Doc No: N2446=07-0316
Date: 2007-10-04
Author: Pablo Halpern
Bloomberg, L.P.

phalpern@halpernwightsoftware.com

The Scoped Allocator Model

Contents

Introduction 1
Document Conventions 2
Motivation 2
Summary of Changes 3
Proposed Wording 4
  Requirements 4
  Allocator-related Type Traits 5
  Pair changes 7
  Container Requirements 9
  basic_string Changes 15
  deque changes 15
  list changes 16
  vector changes 16
  Changes to adapters 17
  map changes 18
  multimap changes 19
  set changes 19
  multset changes 20
  unordered_map changes 20
  unordered_multimap changes 20
  unordered_set changes 21
  unordered_multiset changes 21
Implementation Experience 22

Introduction

This document comprises part 3 of n2387, *Omnibus Allocator Fix-up Proposals*. Inclusion of this section of n2383 into the working paper was approved by the Library Working Group during the morning session of October 4, 2007 in Kona. The rest of n2387 is still on the table, neither approved nor rejected by the LWG.

Document Conventions

All section names and numbers are relative to the August 2007 working draft, N2369.

> Existing and proposed working paper text is indented and shown in dark blue. Small edits to the working paper are shown with ~~green 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 appears with light (yellow) shading.

Motivation

When allocators are allowed to have state, it is necessary to have a model for determining from where an object obtains its allocator. We've identified two such models: the "Moves with Value" allocator model and the "Scoped" allocator model.

In the "Moves with Value" allocator model, the copy constructor of an allocator-aware class will copy both the value *and* the allocator from its argument. This is the model specified in the C++03 standard. With this model, inserting an object into a container usually causes the new container item to copy the allocator from the object that was inserted. This model can be useful in special circumstances, e.g., if the items within a container use an allocator that is specially tuned to the item's type.

In the "Scoped" allocator model, the allocator used to construct an object is determined by the context of that object, much like a storage class. With this model, inserting an object into a container causes the new container item to use the same allocator as the *container*. To avoid allocators being used in the wrong context, the allocator is *never* copied during copy or move construction. Thus, it is possible using this model to use allocators based on short-lived resources without fear that an object will transfer its allocator to a copy that might outlive the (shared) allocator resource. This model is reasonably safe and generally useful on a large scale. There was strong support in the 2005 Tremblant meeting for pursuing an allocator model that propagates allocators from container to contained objects.

With this proposal, we strive to support *both* models well. As we'll see in subsequent sections, clarifying the allocator models allows us to reason about the best solutions to a number of known issues. Note that stateless allocators work identically in both models.

Summary of Changes

The proposed wording for this section is long because similar changes are made in many places in the working draft. The basic concepts can be explained much more concisely, however, and are summarized here.

We begin with two new traits:

```
uses_scoped_allocator<T>
suggest_scoped_allocator<Alloc>
```

Both traits are *elective*, meaning they do not specify an intrinsic quality of the type but rather a deliberate choice by the author of the type. The first trait is specialized for a given type, `T` to derive from `true_type` if `T` uses an allocator and conforms to the "Scoped" allocator model. The second trait is specialized for an allocator type to indicate that client's of that allocator should use the "Scoped" allocator model. All of the standard containers define the first trait if the second trait is true for their `allocator_type`. The class template, `function<F>` also defines the `uses_scoped_allocator` as true.

Every container class, `C`, is enhanced with an *extended move constructor* and *extended copy constructor* as follows:

```
C(C&&, const allocator_type&);       // extended move constructor
C(const C&, const allocator_type&); // extended copy constructor
```

The normal move and copy constructors for each container class are modified to have the following behavior:

> If `uses_scoped_allocator<C>::value` is `true`, then `C(`*`other`*`)` behaves like `C(`*`other`*`, C::allocator_type())`, otherwise `C(`*`other`*`)` behaves like `C(`*`other`*`, other.get_allocator())`.

In other words, if an allocator is not provided to the copy constructor, then the copy constructor behaves differently depending on whether or not the `uses_scoped_allocator` trait is true. If the trait is true, the object uses the default-constructed allocator, otherwise, you get the C++03 behavior and the allocator is copied from the argument.

For each insertion function (including `insert`, `push_back`, `push_front`, and constructors that insert), the following rule is used when copying each inserted value, `v`, into the container:

If `uses_scoped_allocator` is true for both the container and its `value_type`, and if `C::value_type` is constructible with `C::allocator_type` then construct a copy of `v` by calling `C::value_type(v, c.get_allocator())`, i.e., use the *extended copy constructor* or *extended move constructor* for `value_type`. Otherwise, call the normal copy or move constructor, `C::value_type(v)`.

In other words, pass the container's allocator to the constructor of each of the container's elements (if the correct traits are defined and the allocators are compatible).

Class template `pair` is not technically a container, but it must allow its members to be constructed with specific allocators. This proposal adds an allocator argument to each of `pair`'s constructors if either or both of the `pair` member types use the "scoped" allocator model.

Because, depending on the allocator model, allocators are not always copied at copy-construction, it will also be necessary to add allocators to `queue, priority_queue,` and `stack`. The `stringstream` class can also benefit from user-controlled allocation.

Proposed Wording

Requirements

Modify the first paragraph of [utility.arg.requirements] (20.1.1) as follows:

The template definitions in the C++ Standard Library refer to various named requirements whose details are set out in tables 31–~~38~~[new table number]. In these tables, T is a type to be supplied by a C++ program instantiating a template; a, b, and c are values of type const T; s and t are modifiable lvalues of type T; u is a value of type (possibly const) T; ~~and~~ rv is a non-const rvalue of type T, M is a storage allocator type (20.1.2) used by T, and m is a value of type convertible to M.

In section [utility.arg.requirements] (20.1.1), after tables 38, add three more tables:

Table 38+1: `ExtendedDefaultConstructible` requirements

expression	post-condition
T t(m);	`t` uses a copy of `m` to allocate memory.

The constructor for T accepting a single allocator argument is known as the *extended default constructor.*

Table 38+2: `ExtendedMoveConstructible` requirements

expression	post-condition
T t(rv, m);	`t` is equivalent to the value of `rv`. `t` uses a copy of `m` to allocate memory.

[*Note:* This is a *binary* requirement on the *relationship* between `T` and `M`. – *end note*] [*Note:* There is no requirement on the value of `rv` after the assignment. – *end note*]

The constructor for T accepting a `T&&` argument and an allocator argument is known as the *extended move constructor.*

Table 38+3: `ExtendedCopyConstructible` requirements

expression	post-condition
T t(u, m);	The value of `u` is unchanged and is equivalent to `t`. `t` uses a copy of `m` to allocate memory.
[*Note:* This is a *binary* requirement on the *relationship* between `T` and `M`. – *end note*] [*Note:* A pair of types that satisfy the `ExtendedCopyConstructible` requirements also satisfies the `ExtendedMoveConstructible` requirements – *end note*]	

The constructor for T accepting a `const T&` argument and an allocator argument is known as the *extended copy constructor*.

These requirements are needed to describe the requirements and behavior of containers that propagate their own allocator to their contained items (see the `uses_scoped_allocator` trait, below). Like other requirements in this section of the working draft, these new requirements will eventually be implemented as concepts.

Allocator-related Type Traits

In section [memory] (20.6), insert the following class declarations at the *beginning* of the **Header `<memory>` synopsis**:

```
// 2.6.x, allocator-related traits
template <class T> struct uses_scoped_allocator;
template <class Alloc> struct suggest_scoped_allocator;
template <class T, class Alloc> struct constructible_with_allocator;
```

Insert before [default.allocator] (20.6.1):

2.6.x Allocator-related traits [allocator.traits]

The class templates, `uses_scoped_allocator` and `suggest_scoped_allocator` meet the *UnaryTypeTrait* requirements ([meta.rqmts] 20.4.1). The class template `constructible_with_allocator` meets the requirements of a *BinaryTypeTrait* ([meta.rqmts] 20.4.1). Each of these templates shall be publicly derived directly or indirectly from `true_type` if the corresponding condition is true, otherwise from `false_type`. All are *elective* traits; they are not computed automatically by determining an intrinsic quality of the type but rather indicate a deliberate choice by the author of the type. A program may specialize these traits for user-defined types to indicate that the “Scoped” allocator model is used for a those types. The main attributes of a class that conforms to the “Scoped” allocator model are:

– An object’s allocator is not copied or moved on copy construction or move construction.

– If the class is MoveConstructible or CopyConstructible, then it is also ExtendedMoveConstructible or ExtendedCopyConstructible, respectively ([utility.arg.requirements] 20.1.1).

A conforming container containing items of a class that conforms to the "Scoped" allocator model will pass a copy of the container's allocator to the constructors of the items that it manages.

In table [new table number], `T` denotes any type and `Alloc` denotes a storage allocator, as defined in [allocator.requirements] (20.1.2).

Table [new table number]: Allocator-related traits

Template	**Condition**	**default**
`template <class T>` `uses_scoped_allocator`	T conforms to the "scoped" allocator model.	false
`template <class Alloc>` `suggest_scoped_allocator`	Classes that use `Alloc` should adhere to the "scoped" allocator model	false
`template <class T, class Alloc>` `constructible_with_allocator`	ExtendedDefaultConstructible<T,A> or ExtendedMoveConstructible<T,A>	Note A

Note A: The generic implementation of `constructible_with_allocator` is derived from `true_type` iff T `uses_scoped_allocator<T>::value` and `is_convertible<Alloc, T::allocator_type>::value` are both true. This class must be specialized for any class for which `uses_scoped_allocator<T>::value` is true but which does not have an `allocator_type` member type (e.g. class template `function`, `([func.wrap.func] 20.5.14.2))`. Implementations are permitted to implement this trait in a more sophisticated (and possibly implementation-dependent) way that more accurately detects the actual condition that T is constructible from `Alloc` is the last argument to at least one constructor of T.

Once concepts are finalized, the `uses_scoped_allocator` trait should be computed automatically for most types by detecting the `ExtendedMoveConstructible<T, A>` concept. However, the trait is still needed so that it can be specialized to evaluate false in the case where heuristic detection yields the wrong value. The `constructible_with_allocator` trait, however, can be fully replaced by a using concepts, once they become generally available in compilers.

The `suggest_scoped_allocator` trait provides a "master switch" by which an allocator can select the allocator-model for all of the standard containers and any other container that follows the suggestion. The other alternative we considered was to add an additional (defaulted) template parameter specifying the allocator model for each container type, but that would make the use of the new model very tedious and somewhat error prone.

Note that detecting "constructible with allocator" is difficult in the most general case, even with concepts. We might want to require that all allocator-aware classes supply an allocator_type declaration, even those classes, like function, that use type-erasure. In the type-erasure case, allocator_type would be some generic type that is convertible from any allocator type. This might replace one complexity with another. Opinions?

Pair changes

In section [pairs] (20.2.3), add a new paragraph after paragraph 1:

> A pair can be instantiated on almost any two types, provided the first type can be constructed with zero or one argument. [Is this correct?] If either or both of the types uses a storage allocator ([allocator.requirements] 20.1.2) and has the uses_scoped_allocator trait, then the instantiated pair class also uses an allocator and uses_scoped_allocator is specialized to true_type for the pair. An allocator passed as an extra argument to a pair constructor will be passed on to one or both of the pair's elements, provided that it is compatible with that element's allocator.

Then, modify the declaration of pair<T1, T2>, as follows:

```
template <class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;
    pair();
    pair(const T1& x , const T2& y );
    template<class U , class V > pair(U&& x , V&& y );
    pair(pair&& p );
    template<class U , class V > pair(const pair<U , V >& p );
    template<class U, class... Args> pair(U&& x, Args&&... args);

    template <class Alloc> pair(const Alloc& a);
    template <class Alloc>
      pair(const T1& x, const T2& y, const Alloc& a);
    template<class U , class V, class Alloc >
      pair(U&& x , V&& y const Alloc& a);
    template <class Alloc> pair(pair&& p, const Alloc& a);
    template<class U , class V, class Alloc >
      pair(const pair<U , V >& p, const Alloc& a );
    template<class U , class V, class Alloc >
      pair(pair<U, V>&& p, const Alloc& a );

    pair& operator=(pair&& p );
    template<class U , class V > pair& operator=(pair<U , V >&& p );

    void swap(pair&& p );
};
```

After the definition of `template<class U,class V> pair(pair<U,V >&& p ),` add the following definitions:

```
template <class Alloc> pair(const Alloc& a);
template <class Alloc>
  pair(const T1& x, const T2& y, const Alloc& a);
template<class U , class V, class Alloc >
  pair(U&& x , V&& y const Alloc& a);
template <class Alloc> pair(pair&& p, const Alloc& a);
template<class U , class V, class Alloc >
  pair(const pair<U , V >& p, const Alloc& a );
template<class U , class V, class Alloc >
  pair(pair<U, V>&& p, const Alloc& a );
```

requires: `Alloc` shall be an `Allocator` ([allocator.requirements] 20.1.2); `uses_scoped_allocator<pair>` (*see below*); `constructible_with_allocator<pair, Alloc>` (*see below*).

effects: equivalent to the previous six constructors except that the allocator argument is passed conditionally to the constructors of `first`, `second`, or both. If `uses_scoped_allocator<T1>::value && constructible_with_allocator<T1,Alloc>::value`, the `a` is passed as the last argument to the constructor for `first`. Similarly, if `uses_scoped_allocator<T2>::value && constructible_with_allocator<T2,Alloc>::value`, the `a` is passed as the last argument to the constructor for `second`.

These definitions allow containers (especially associative containers) to pass an allocator to items of `pair` type. There are probably ambiguities created by these additional definitions. These ambiguities can be eliminated by combining ambiguous constructors into a single prototype, then using meta-programming to distinguish an allocator argument from a normal argument. Once `Allocator` is implemented as a concept, the ambiguities should disappear.

```
template <class T1, class T2>
struct uses_scoped_allocator<pair<T1, T2> > : see below;
```

Derived directly or indirectly from `true_type` if `uses_scoped_allocator<T1>::value || uses_scoped_allocator<T2>::value`, else derived directly or indirectly from `false_type`.

```
template <class T1, class T2, class Alloc>
struct constructible_with_allocator<pair<T1, T2>, Alloc> : see below;
```

requires: `Alloc` shall be an `Allocator` ([allocator.requirements] 20.1.2)

Derived directly or indirectly from `true_type` if `constructible_with_allocator<T1, Alloc>::value || constructible_with_allocator<T2, Alloc>::value`, else derived directly or indirectly from `false_type`.

Automatically determine `pair` traits based on the traits of its elements.

Note that something similar to the changes above would also be needed for `tuple`.

Container Requirements

Reword [container.requirements] (23.1), paragraph 8 as follows:

> ~~Copy constructors for all container types defined in this clause copy an allocator argument from their respective first parameters.~~ All ~~other~~ constructors except the copy and move constructors for ~~these~~ the container types defined in this clause take a~~n~~ const 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. In all container types defined in this clause, the member get_allocator() returns a copy of the Allocator object used to construct the container.~~253)~~
>
> The allocator selected by a container during move construction or copy construction depends on the allocator model, as set by the value of the `uses_scoped_allocator` trait for the container. If the trait is false, the move and copy constructors copy the allocator from their argument. If the trait is true, then the allocator is default-constructed. [*Note:* if the trait is used and the allocator type is not DefaultConstructible, then the container will not be MoveConstructible or CopyConstructible (though it could still be ExtendedMoveConstructible and ExtendedCopyConstructible). – *end note*]
>
> ~~253) As specified in 20.1.2, paragraphs 4-5, the semantics described in this clause applies only to the case where allocators compare equal.~~

The trait-based copy/move semantics prevent allocators from being transferred on copy and move construction when the "Scoped" allocator model is in use.

The behavior and performance of move and copy constructors is unchanged for stateless allocators and for the (common) case where the object being moved has an allocator equal to the default-constructed allocator. Otherwise, the move constructor will become an O(n) operation instead of an O(1) operation. In the spirit of "you pay only for what you use," only users who care about using multiple, distinct values of stateful allocators with the new model will pay this penalty, and even they can avoid the penalty under most circumstances. Also, in the spirit of "support the novice without interfering with the expert," the default behavior is safe and consistent with the model, and an experienced allocator-user can pass the allocator explicitly in such a way as to ensure that the move is fast.

In [memory] (20.6), before the declaration of `uninitialized_copy`, add the following algorithm declaration:

```
template <class C>
  typename C::allocator_type
    select_allocator_for_copy(const C&);

template <class C, class Alloc>
  typename Alloc select_allocator_for_copy(const C&, Alloc&& A);
```

In [specialized.algorithms], before [uninitialized.copy] (2.6.4.1) insert:

2.6.4.y template function `select_allocator_for_copy` **[select.allocator]**

```
template <class C>
  typename C::allocator_type
    select_allocator_for_copy(const C& container);
```

Requires: `C` provides a type `allocator_type` and a member function, `get_allocator()` that returns `allocator_type`. A program is permitted to overload this function for user-defined classes.

Returns: If `uses_scoped_allocator<C >`, then returns `C::allocator_type()`, otherwise returns `container.get_allocator()`.

```
template <class C, class Alloc>
  typename Alloc select_allocator_for_copy(const C&, Alloc&& A);
```

A program is permitted to overload this function for user-defined classes.

Requires: C has a member type, `allocator_type`.

Returns: If `uses_scoped_allocator<C>`, then returns `Alloc(C::allocator_type())`, otherwise returns `Alloc(move(A))`.

These are helpful functions for implementing the semantics of copy and move construction for containers as described above.

In section [container.requirements] (23.1), replace paragraph 3:

~~Objects stored in these components shall be MoveConstructible and MoveAssignable. If the copy constructor of a container is used, objects stored in that container shall be CopyConstructible. If the copy assignment operator of a sequence container is used, objects stored in that container shall be CopyConstructible and CopyAssignable. If the copy assignment operator of an associative container is used, objects stored in that container shall be CopyConstructible.~~

For a container C, using allocator A and containing items of type T, if
`items_use_containers_allocator<C>::value &&`
`items_use_containers_allocator<T>::value &&`

`consructible_with_allocator<T,A>::value`, then the container will pass its allocator as an additional argument to T's constructor for each of the container's items. In this case, the requirements on T in all of the tables in this clause (including Tables 87, 89, 90, 91, and 93) are modified such that MoveConstructible is replaced by ExtendedMoveConstructible, CopyConstructible is replaced by ExtendedCopyConstructible, and DefaultConstructible is replaced by ExtendedDefaultConstructible (with respect to the container's allocator).

The requirements on T should be stated on a per-function basis in the tables, to avoid unnecessary restrictions. For example there is no need for T to be MoveAssignable if a function that uses move-assignment is never invoked. The `uses_scoped_allocator` trait is used to choose the allocator model. The allocator is propagated from the container to the contained item if and only if both the container and the item agree to this contract. If they do agree, the container passes its own allocator to the item when it constructs the item. The use of the model is determined once for the container; it does not vary from function to function, e.g., the container will not propagate the allocator on, say, move construction but not on copy construction. Note that this paragraph does not require that either the container or the item type use an allocator (because allocator-specific behavior depends on the `uses_scoped_allocator` trait, which applies only to classes that use allocators).

In section [container.requirements] (23.1), Table 87: Container requirements, change selected rows as follows:

expression	return type	operational semantics	assertion/note pre/post-condition	complexity
`X::value_-type`	`T`		~~T is `CopyConstructible`~~	compile time
…				
`X(a);`			*requires*: T is CopyConstructible. `a == X(a)`	linear
`X u(a);` `X u = a;`			*requires*: T is CopyConstructible. post: `u == a` ~~Equivalent to: X u; u = a;~~	linear
`X u(rv);` `X u = rv;`			*requires*: T is MoveConstructible. post: u shall be equal to the value that rv had before this construction ~~Equivalent to: X u; u = rv;~~	~~constant~~ (Note B)

Modify the paragraph immediately following Table 87 as follows:

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 worst-case linear complexity, but will often have constant complexity.

In section [container.requirements] (23.1), after paragraph 12 (just before [sequence.reqmts]) add the following text and additional table:

All of the containers defined in this clause and in clause [basic.string] (21.3), except `array`, meet the additional requirements of an allocator-aware container, as described in Table [88+1].

In Table [88+1], X denotes an allocator-aware container class of element type T using allocator type Alloc, u denotes a variable, t denotes an lvalue or a const rvalue of type X, rv denotes a non-const rvalue of type X, m is a value of type Alloc.

Table [88+1] Allocator-aware container requirements (in addition to container)

expression	**return type**	**assertion/note pre/post-condition**	**complexity**
`allocator_type`	`Alloc`	*requires:* `allocator_type::value_type` is the same as `value_type`.	compile time
`uses_scoped_allocator<X>`	derived from `true_type` or `false_type`	true if `suggest_scoped_allocator<Alloc>` is true	compile time
`get_allocator()`	`Alloc`		constant

`X()`
`X u;`
		requires: Alloc is DefaultConstructible.
		`post: X().size() == 0, get_allocator()== Alloc()`
			constant

`X(m)`
`X u(m);`
		`post: a.size() == 0, get_allocator() == m`	constant

`X(t)`
`X u(t);`
		requires: T is CopyConstructible; Alloc is DefaultConstructible.
		`post: u == a`
			linear

`X(t,m)`
`X u(t,m);`
		requires: T is CopyConstructible
		`post: u == a, get_allocator() == m`
			linear

`X(rv)`
`X u(rv);`
		requires: T shall be MoveConstructible
		`post: u == a`
			linear if m != Alloc() and uses_scoped_allocator<X>, else constant

`X(rv,m)`
`X u(rv,m);`
		requires: T shall be MoveConstructible
		`post: u == a, get_allocator() == m`
			constant if m == rv.get_allocator(), else linear

Add the allocator requirements. The `uses_scoped_allocator` traitis computed automatically from `suggest_scoped_allocator`. We specify the extended default, move, and copy constructors, and clarify the complexity of the normal default, move, and copy constructors. Note that *all* containers now have a constructor that takes a single allocator argument. The absence of such a constructor has caused grief for those of us using stateful allocators up until now.

In section [sequence.reqmts] (23.1.1), modify paragraph 3 as follows:

> In Tables 89 and 90, X denotes a sequence container class, a denotes a value of type X containing elements of type T, i and j denote iterators satisfying input iterator requirements and refer to elements implicitly convertible to value_type, [i, j) denotes a valid range, n denotes a value of X::size_type, p denotes a valid const iterator to a, q denotes a valid dereferenceable const iterator to a, [q1, q2) denotes a valid range of const iterators in a, t denotes an lvalue or a const rvalue of X::value_type, and rv denotes a non-const rvalue of X::value_type. Args denotes a template parameter pack; args denotes a function parameter pack with the pattern Args&&.

In section [container.requirements] (23.1), Table 89, change selected rows as follows:

`a.emplace(p,args);`	`iterator`	*requires*: T shall be constructible from args and CopyAssignable. Inserts an object of type T constructed with T(std::forward<Args>(args)...).~~;~~
`a.insert(p,t)`	`iterator`	*requires*: T shall be CopyConstructible and CopyAssignable. inserts a copy of t before p.
`a.insert(p,rv)`	`iterator`	*requires*: T shall be MoveConstructible and MoveAssignable. inserts a copy of rv before p.
`a.erase(q)`	`iterator`	*requires:*T shall be MoveAssignable. Erases the element pointed to by q
`a.erase(q1,q2)`	`iterator`	*requires:*T shall be MoveAssignable. Erases the elements in the range [q1,q2)

In section [sequence.reqmts] (23.1.1), modify rows in Table 90 as follows:

`a.push_-front(args)`	`void`	`a.emplace(a.begin(), std::forward<Args>(args)…)` *requires:* T shall be constructible from args	`list, deque`
`a.push_-back(args)`	`void`	`a.emplace(a.end(), std::forward<Args>(args)…)` *requires:* T shall be constructible from args	`list, deque, vector`

We specify the requirements for `push_front` and `push_back` because they turn out to be less than the requirements for `emplace`.

In section [associative.reqmts] (23.1.2): Associative containers, modify paragraph 2 as follows:

> 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 model in use by the container and the allocator-related traits of the Compare type.

In section [associative.reqmts] (23.1.2): Associative containers, modify paragraph 7 as follows:

In Table 91, X denotes an associative container class, a denotes a value of X, a_uniq denotes a value of X when X supports unique keys, a_eq denotes a value of X when X supports multiple keys, u denotes an identifier, r denotes an lvalue or a const rvalue of type X, and rv denotes a non-const rvalue of type X. i and j satisfy input iterator requirements and refer to elements implicitly convertible to value_type. [i,j) denotes a valid range, p denotes a valid const iterator to a, q denotes a valid dereferenceable const iterator to a, [q1, q2) denotes a valid range of const iterators in a, t denotes a value of X::value_type, k denotes a value of X::key_type and c denotes a value of type X::key_compare. M denotes the storage allocator used by X and m denotes an allocator of type convertible to M.

In section [associative.reqmts] (23.1.2): Associative containers, modify table 91 as follows:

`X(c)`
`X a(c)`
	requires: `key_compare` is CopyConstructible
	constructs an empty container
	uses a copy of c as a comparison object
		constant

`X()`
`X a;`
	requires: `key_compare` is DefaultConstructible
	constructs an empty container
	uses Compare() as a comparison object
		constant

`X(i,j,c)`
`X a(i,j,c);`
	requires: `key_compare` is CopyConstructible
	constructs an empty container and inserts elements from the range `[i,j)` into it; uses a copy of c as a comparison object
		*N*log*N* in general (*N* is the distance from i to j); linear if [i, j) is sorted with value_compare()

`X(i,j)`
`X a(i,j);`
	requires: `key_compare` is DefaultConstructible
	same as above, but uses `Compare()`, as a comparison object.
		same as above

In section [unord.req] (23.1.3), modify paragraph 3 as follows:

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 model in use by the container and the allocator-related traits of the Hash and Pred types.

basic_string Changes

In section [basic.string] (21.3), modify paragraph 3 as follows:

> The class template basic_string conforms to the requirements for a Sequence (23.1.1), ~~and~~ for a Reversible Container (23.1) , and for an allocator-aware container (23.1). T~~hus, t~~he iterators supported by basic_string are random access iterators (24.1.5).

In section [basic.string] (21.3), add the following constructors:

```
basic_string(const basic_string&, const Allocator&);
basic_string(basic_string&&, const Allocator&);
```

In section [basic.string] (21.3), modify the description of the copy and move constructors as follows:

```
basic_string(const basic_string<charT,traits,Allocator>& str);
basic_string(basic_string<charT,traits,Allocator>&& str);
```

> *Effects:* Constructs an object of class `basic_string` as indicated in Table 58. In the first form, the stored Allocator value is ~~copied from str.get_allocator()~~ constructed *as if* copied from `select_allocator_for_copy(str)`. In the second form, the stored Allocator value is ~~move~~ constructed *as if* moved from ~~str.get_allocator()~~ `select_allocator_for_copy(str, move(`*strAlloc*`))`, and str is left in a valid state with an unspecified value.
>
> *Throws*: The second form throws nothing if the allocator's ~~move~~ constructor throws nothing.

Then add descriptions of the extended copy and move constructors:

```
basic_string(const basic_string& str, const Allocator& alloc);
basic_string(basic_string&& str, const Allocator& alloc);
```

> *Effects:* Constructs an object of class `basic_string` as indicated in Table [58+1]. The stored allocator is constructed from `alloc`. In the second form, `str` is left in a valid state with an unspecified value.
>
> *Throws:* The second form throws nothing if `alloc == str.get_allocator()` and the allocator's copy constructor throws nothing.

Element	Value
`data()`	points to the first element of an allocated copy of the array whose first element is pointed at by the original value of `str.data()`
`size()`	the original value of `str.size()`
`capacity()`	a value at least as large as `size()`

deque changes

In section [deque] (23.2.2): Class template `deque`, modify paragraph 2:

A deque satisfies all of the requirements of a container, ~~and~~ of a reversible container, and of an allocator-aware container (given in tables in 23.1) and of a sequence container, including the optional sequence container requirements (23.1.1). Descriptions are provided here only for operations on deque that are not described in one of these tables or for operations where there is additional semantic information.

Add the following constructors:

```
deque(const deque&, const Allocator&);
deque(deque&&, const Allocator&);
```

And add the following trait specialization:

```
template <class T, class Allocator>
struct uses_scoped_allocator<deque<T, Allocator> >
    : suggest_scoped_allocator<Allocator>::type { };
```

list changes

In section [list] (23.2.3): Class template `list`, modify paragraph 2:

A list satisfies all of the requirements of a container, ~~and~~ of a reversible container, and of an allocator-aware container (given in ~~two~~ tables in 23.1) and of a sequence container, including most of the the optional sequence container requirements (23.1.1). The exceptions are the operator[] and at member functions, which are not provided.[258]) Descriptions are provided here only for operations on list that are not described in one of these tables or for operations where there is additional semantic information.

Add the following constructors:

```
list(const list&, const Allocator&);
list(list&&, const Allocator&);
```

And add the following trait specialization:

```
template <class T, class Allocator>
struct uses_scoped_allocator<list<T, Allocator> >
    : suggest_scoped_allocator<Allocator>::type { };
```

vector changes

In section [vector] (23.2.5): Class template `vector`, modify paragraph 2:

A vector satisfies all of the requirements of a container, ~~and~~ of a reversible container, and of an allocator-aware container (given in ~~two~~ tables in 23.1) and of a sequence container, including most of the optional sequence container requirements (23.1.1). The exceptions are the push_front and pop_front member functions, which are not provided. Descriptions are provided here only for operations on vector that are not described in one of these tables or for operations where there is additional semantic information.

Add the following constructors:

```
vector(const vector&, const Allocator&);
vector(vector&&, const Allocator&);
```

And add the following trait specialization:

```
template <class T, class Allocator>
struct uses_scoped_allocator<vector<T, Allocator> >
    : suggest_scoped_allocator<Allocator>::type { };
```

In section [vector.bool] (23.2.6): Class `vector<bool>`, add the following constructors:

```
vector(const vector&, const Allocator&);
vector(vector&&, const Allocator&);
```

No additional specialization of `uses_scoped_allocator` is needed for `vector<bool>`. The specialization for `vector<T>` is sufficient.

Changes to adapters

In section [container.adaptors] (23.2.4): Container adaptors, modify paragraph 1 as follows:

> The container adaptors each take a Container template parameter, and each constructor takes a Container reference argument. This container is copied into the Container member of each adaptor. If the container takes an allocator, then a compatible allocator may be passed in to the adaptor's constructor. Otherwise, normal copy or move construction is used for the container argument. [*Note:* it is not necessary for an implementation to distinguish between the one-argument constructor that takes a `Container` and the one-argument constructor that takes an `allocator_type`. Both forms use their argument to construct an instance of the container. – *end note*]

If a container adheres to the "Scoped" allocator model, there is no other way to specify the allocator to be used by the copy of the container within the adapter. As all of the proposals in this paper are about making allocators more useful, it is reasonable that we make it easy to specify allocators ubiquitously.

In section [queue.defn] (23.2.4.1.1): `queue` definition, add the following constructors:

```
template <class Alloc> explicit queue(const Alloc&);
template <class Alloc> queue(const Container&, const Alloc&);
template <class Alloc> queue(Container&&, const Alloc&);
template <class Alloc> queue(queue&&, const Alloc&);
```

And add the following trait specialization:

```
template <class T, class Container>
struct uses_scoped_allocator<queue<T, container> >
    : uses_scoped_allocator<Container>::type { };

template <class T, class Container, class Alloc>
struct constructible_with_allocator<queue<T, container>, Alloc >
    : constructible_with_allocator<Container, Alloc>::type { };
```

In section [priority.queue] (23.2.4.2): Class template `priority_queue`, add the following constructors:

```
template <class Alloc> explicit priority_queue(const Alloc&);
template <class Alloc> priority_queue(const Container&,
                                      const Alloc&);
template <class Alloc> priority_queue(Container&&,
                                      const Alloc&);
template <class Alloc> priority_queue(priority_queue&&,
                                      const Alloc&);
```

And add the following trait specializations:

```
template <class T, class Container>
struct uses_scoped_allocator<priority_queue<T, container> >
    : uses_scoped_allocator<Container>::type { };

template <class T, class Container, class Alloc> struct
constructible_with_allocator<priority_queue<T, container>, Alloc >
    : constructible_with_allocator<Container, Alloc>::type { };
```

In section [stack.defn] (23.2.4.3.1): `stack` definition, add the following constructors:

```
template <class Alloc> explicit stack(const Alloc&);
template <class Alloc> stack(const Container&, const Alloc&);
template <class Alloc> stack(Container&&, const Alloc&);
template <class Alloc> stack(stack&&, const Alloc&);
```

And add the following trait specializations:

```
template <class T, class Container>
struct uses_scoped_allocator<stack<T, container> >
    : uses_scoped_allocator<Container>::type { };

template <class T, class Container, class Alloc>
struct constructible_with_allocator<stack<T, container>, Alloc >
    : constructible_with_allocator<Container, Alloc>::type { };
```

map changes

In section [map] (23.3.1): Class template map, change paragraph 2 as follows:

A map satisfies all of the requirements of a container and of a reversible container (23.1), of an allocator-aware container (23.1), and of an associative container (23.1.2). A map also provides most operations described in (23.1.2) for unique keys. This means that a map supports the a_uniq operations in (23.1.2) but not the a_eq operations. For a map<Key,T> the key_type is Key and the value_- type is pair<const Key,T>. Descriptions are provided here only for operations on map that are not described in one of those tables or for operations where there is additional semantic information.

Add the following constructors:

```
map(const Allocator&);
map(const map&, const Allocator&);
map(map&&, const Allocator&);
```

And add the following trait specialization:

```
template <class Key, class T, class Compare, class Allocator>
struct uses_scoped_allocator<map<Key,T,Compare,Allocator> >
    : suggest_scoped_allocator<Allocator>::type { };
```

multimap changes

In section [multimap] (23.3.2): Class template multimap, change paragraph 2 as follows:

A multimap satisfies all of the requirements of a container and of a reversible container (23.1), of an allocator-aware container (23.1), and of an associative container (23.1.2). A multimap also provides most operations described in (23.1.2) for equal keys. This means that a multimap supports the a_eq operations in (23.1.2) but not the a_uniq operations. For a multimap<Key,T> the key_type is Key and the value_type is pair<const Key,T>. Descriptions are provided here only for operations on multimap that are not described in one of those tables or for operations where there is additional semantic information.

And add the following trait specialization:

```
template <class Key, class T, class Compare, class Allocator> struct
uses_scoped_allocator<multimap<Key,T,Compare,Allocator> >
    : suggest_scoped_allocator<Allocator>::type { };
```

set changes

In section [set] (23.3.3) Class template set, change paragraph 2 as follows:

A set satisfies all of the requirements of a container and of a reversible container (23.1), of an allocator-aware container (23.1), and of an associative container (23.1.2). A set also provides most operations described in (23.1.2) for unique keys. This means that a set supports the a_uniq operations in (23.1.2) but not the a_eq operations. For a set<Key> both the key_type and value_type are Key. Descriptions are provided here only for operations on set that are not described in one of these tables and for operations where there is additional semantic information.

And add the following trait specialization:

```
template <class Key, class Compare, class Allocator> struct
uses_scoped_allocator<set<Key,Compare,Allocator> >
    : suggest_scoped_allocator<Allocator>::type { };
```

multset changes

In section [multiset] (23.3.4): Class template multiset, modify paragraph 2 as follows:

> A multiset satisfies all of the requirements of a container and of a reversible container (23.1), of an allocator-aware container (23.1), and of an associative container (23.1.2). multiset also provides most operations described in (23.1.2) for duplicate keys. This means that a multiset supports the a_eq operations in (23.1.2) but not the a_uniq operations. For a multiset<Key> both the key_type and value_type are Key. Descriptions are provided here only for operations on multiset that are not described in one of these tables and for operations where there is additional semantic information.

And add the following trait specialization:

```
template <class Key, class Compare, class Allocator> struct
uses_scoped_allocator<multiset<Key,Compare,Allocator> >
    : suggest_scoped_allocator<Allocator>::type { };
```

unordered_map changes

In section [unord.map] (23.4.1): Class template unordered_map, modify paragraph 2 as follows:

> An unordered_map satisfies all of the requirements of a container, of an allocator-aware container, and of an unordered associative container. It provides the operations described in the preceding requirements table for unique keys; that is, an unordered_map supports the a_uniq operations in that table, not the a_eq operations. For an unordered_map<Key, T> the key type is Key, the mapped type is T, and the value type is std::pair<const Key, T>.

And add the following trait specialization:

```
template <class Key,class T,class Hash,class Pred,class Allocator>
struct uses_scoped_allocator<unordered_map<
                                     Key,T,Hash,Pred,Allocator> >
    : suggest_scoped_allocator<Allocator>::type { };
```

unordered_multimap changes

In section [unord.multimap] (23.4.2): Class template unordered_multimap, modify paragraph 2 as follows:

> An unordered_multimap satisfies all of the requirements of a container, of an allocator-aware container, and of an unordered associative container. It provides the operations described in the preceding requirements table for equivalent keys; that is, an unordered_- multimap supports the a_eq operations in that table, not the a_uniq operations. For an unordered_multimap<Key, T> the key type is Key, the mapped type is T, and the value type is std::pair<const Key, T>.

And add the following trait specialization:

```
template <class Key,class T,class Hash,class Pred,class Allocator>
struct uses_scoped_allocator<unordered_multimap<
                                     Key,T,Hash,Pred,Allocator> >
    : suggest_scoped_allocator<Allocator>::type { };
```

unordered_set changes

In section [unord.set] (23.4.3): Class template unordered_set, modify paragraph 2 as follows:

> An unordered_set satisfies all of the requirements of a container, of an allocator-aware container, and of an unordered associative container. It provides the operations described in the preceding requirements table for unique keys; that is, an unordered_set supports the a_uniq operations in that table, not the a_eq operations. For an unordered_set<Value> the key type and the value type are both Value. The iterator and const_iterator types are both const iterator types. It is unspecified whether they are the same type.

And add the following trait specialization:

```
template <class Value,class Hash,class Pred,class Allocator>
struct uses_scoped_allocator<unordered_set<
                                     Value,Hash,Pred,Allocator> >
    : suggest_scoped_allocator<Allocator>::type { };
```

unordered_multiset changes

In section [unord.set] (23.4.3): Class template unordered_multiset, modify paragraph 2 as follows:

> An unordered_multiset satisfies all of the requirements of a container, of an allocator-aware container, and of an unordered associative container. It provides the operations described in the preceding requirements table for equivalent keys; that is, an unordered_multiset supports the a_eq operations in that table, not the a_uniq operations. For an unordered_multiset<Value> the key type and the value type are both Value. The iterator and const_iterator types are both const iterator types. It is unspecified whether they are the same type.

And add the following trait specialization:

```
template <class Value,class Hash,class Pred,class Allocator>
struct uses_scoped_allocator<unordered_multiset<
                                     Value,Hash,Pred,Allocator> >
    : suggest_scoped_allocator<Allocator>::type { };
```

Implementation Experience

Most of the elements in this section have been implemented and used extensively at Bloomberg for several years. We make frequent use of short-lived arena allocators and allocators that use special memory regions, and these semantics have provided a powerful way to manage memory. There is also a second implementation by a commercial vendor.