Doc No: N2387=07-0247
Date: 2007-09-10
Author: Pablo Halpern
Bloomberg, L.P.

phalpern@halpernwightsoftware.com

Omnibus Allocator Fix-up Proposals

Contents

Introduction 2
Document Conventions 2
1. Remove Weasel Words 3
2. Library-wide Requirements for Use of Allocators 4
3. The "Scoped" Allocator Model 5
  Requirements 7
  Allocator-related Type Traits 8
  Pair changes 10
  Container Requirements 12
  basic_string Changes 18
  deque changes 18
  list changes 19
  vector changes 19
  Changes to adapters 20
  map changes 21
  multimap changes 22
  set changes 22
  multset changes 23
  unordered_map changes 23
  unordered_multimap changes 23
  unordered_set changes 24
  unordered_multiset changes 24
4. Polymorphic Allocators 25
5. Polymorphic Allocator as the Default Allocator 33
6. Semantics of `swap` 34
7. Semantics of `pointer` and `address` in Allocators 35
8. Allow the Use of `address` for Foreign Objects 37
9. Consistent Copy and Equality Semantics for Allocators 38
10. Add variadic `construct` Requirement for Allocators 39

11. Correction to `function` interface 40
12. Allocator-aware stringstream 40
13. Acknowledgements 40

Introduction

This series of proposals is intended to address numerous defect reports and enhancement requests related to allocators in the standard library. To be optimally useful, allocators must conform to a well-defined model whereby library facilities always allocate memory from an optional user-supplied allocator.

These proposals address the following issues:

- LWG 580: must use construct, destroy, address
- N1850: Towards a Better Allocator Model
- LWG 401: incorrect type casts in table 32 in lib.allocator.requirements
- LWG 634: turn `address` into `boost::addressof`
- LWG 635: domain of `allocator::address`
- Make templated rvalue-ref `construct` variadic.
- Create concepts for a pointer type
- LWG 258: Missing allocator requirement (transitive ==)
- LWG 431 (N1599): Swapping containers with unequal allocators
- Allocator copy issues with `std::function`
- Require that `Container::value_type` match `Container::allocator_type::value_type`.

Because concepts are not yet final, I have deferred most concept-related issues in this paper. Concepts will play an important roll in allocator usage, however, and I have tried to point out where I think concepts can be applied.

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.

1. Remove Weasel Words

Motivation

The 1998 standard contains words that leave several important details of allocator usage to the implementation. These vagaries prevent the portable use of stateful allocators and allocators that use an unconventional memory model.

Proposed Wording

In section [allocator.requirements] (20.1.2), remove the last paragraphs 4 and 5:

~~Implementations of containers described in this International Standard are permitted to assume that their Allocator template parameter meets the following two additional requirements beyond those in Table 40.~~

- ~~All instances of a given allocator type are required to be interchangeable and always compare equal to each other.~~
- ~~The typedef members pointer, const_pointer, size_type, and difference_type are required to be T*, T const*, std::size_t, and std::ptrdiff_t, respectively.~~

~~Implementors are encouraged to supply libraries that can accept allocators that encapsulate more general memory models and that support non-equal instances. In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in Table 40, and the semantics of containers and algorithms when allocator instances compare non-equal, are implementation-defined.~~

Removing these words is a prerequisite for the other proposals in this paper. This paper proposes standard behavior for stateful allocators and allocators that use smart pointers.

2. Library-wide Requirements for Use of Allocators

Motivation

As described in LWG Issue 580, the 1998 standard specifies requirements for allocators, including required member functions and types, but does not require that the standard library used those members. If those facilities are not used (e.g., the `construct` function is not called), then the author of the allocator cannot properly control the way memory is used. This is especially problematic when the allocator provides special pointer and reference types.

Proposed Wording

At the beginning of section [conforming] (17.4.4), change the introductory paragraph as follows:

> **17.4.4 Conforming implementations [conforming]**
>
> This subclause describes the constraints upon, and latitude of, implementations of the C++ Standard library. The following subclauses describe an implementation's use of headers (17.4.4.1), macros (17.4.4.2), global functions (17.4.4.3), member functions (17.4.4.4), reentrancy (17.4.4.5), access specifiers (17.4.4.6), class derivation (17.4.4.7), ~~and~~ exceptions (17.4.4.8), and allocators (17.4.4.9).

After [res.on.exception.handling] (17.4.4.8), insert a new section:

> **17.4.4.9 Use of Allocators [use.of.allocators]**
>
> Many of the classes (including instantiations of class templates) defined in the C++ Standard Library are constructed with a user-supplied object that meets the requirements for a memory allocator ([allocator.requirements] 20.1.2). A copy of this allocator shall be used, by the constructors and by all member functions of standard library classes, to allocate, construct, destroy, deallocate, and obtain pointers to objects whose lifetime is managed by the class object, including but not limited to those of a container's `value_type`. Allocation shall be performed "as if" by calling the allocate() member function on a copy of the allocator object of the appropriate type [New Footnote], and deallocation "as if" by calling deallocate() on a copy of the same allocator object of the corresponding type. All objects residing in storage allocated by a container's allocator shall be constructed "as if" by calling the construct() member function on a copy of the allocator object of the appropriate type. The same objects shall be destroyed "as if" by calling destroy() on a copy of the same allocator object of the same type. The address of such objects shall be stored within the object using the allocator's `pointer` or `const_pointer` types and obtained "as if" by calling the address() member function on a copy of the allocator object of the appropriate type. For classes that define a max_size() member function, the value returned from max_size() shall be no larger than the value returned by calling max_size() on a copy the object's allocator.
>
> **New Footnote:** This type may be different from Allocator: it may be derived from Allocator via Allocator::rebind<U>::other for the appropriate type U.

This description was chosen so that the allocator author would have maximal control over how memory is used from within a library object, especially if non-standard pointers (e.g. a smart pointers into special memory) are used. The intent is to require that the allocator be used for objects that are part of the data structure, but not for temporaries that are not managed by the class object. The wording is largely the same as Martin Sebor's proposed resolution for LWG 580. However, because there are now non-container classes in the library that use allocators (e.g. `shared_ptr` and `function`), I have moved the wording up from the container requirements section to the library-wide requirements section and have made the wording less container-centric (though one mention of containers was necessary). Also, since some of the new uses of allocator use type-erasure and do not parameterize the class on the allocator, I have removed references to such parameterization.

3. The "Scoped" Allocator Model

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 "item use container's allocator" model.	false
`template <class Alloc>` `suggest_scoped_allocator`	Classes that use `Alloc` should adhere to the "item use container's 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. By the time we meet in Kona, there will be at least a second implementation, this time by a commercial vendor.

4. Polymorphic Allocators

Motivation

One of the most common difficulties people have in using custom allocators is that the allocator type is part of the container type. Thus, `std::vector<int,MyAlloc>` is a different type then `std::vector<int,YourAlloc>`. This prevents the former from being passed to a function that expects the latter, even as a `const` reference. A stateful allocator can be constructed that is essentially a wrapper around a pointer to an abstract allocation mechanism. The actual allocation mechanism used by any particular object would be determined at run-time, and would not affect the *type* of the object. The "Scoped" allocator model would prevent such an allocator from accidentally ending up in the wrong place.

A polymorphic allocator class is most useful if it is standardized so that everybody is encouraged to use the same one, thus maximizing interoperability among modules. We propose such a class here, along with an adapter that allows almost any allocator to be used in the polymorphic context. This proposal assumes acceptance of the "Items Use Container Allocator" Model proposal.

Proposed Wording

In section [memory] (20.6), before the definition of the default allocator, insert:

```
class allocator_mechanism;

template <class T> class polymorphic_allocator;
template <> class polymorphic_allocator<void>;
template <class T> struct
  suggest_scoped_allocator< polymorphic_allocator<T> >;
template <class T, class U>
  bool operator==(const polymorphic_allocator<T>&,
                  const polymorphic_allocator<U>&) throw();
template <class T, class U>
  bool operator!=(const polymorphic_allocator<T>&,
                  const polymorphic_allocator<U>&) throw();
```

```
class new_delete_allocator_mechanism;

template <typename Allocator> class allocator_mechanism_adapter;

allocator_mechanism*
  set_default_allocator_mechanism(allocator_mechanism* m);
allocator_mechanism*
  set_global_allocator_mechanism(allocator_mechanism* m);

allocator_mechanism* default_allocator_mechanism();
allocator_mechanism* global_allocator_mechanism();
```

Before [default.allocator] (20.6.1), add the following sections:

20.6.c The polymorphic allocator mechanism [allocator.mechanism]

Class `allocator_mechanism` is an abstract base class defining a polymorphic memory allocation protocol.

```
namespace std {
  class allocator_mechanism
  {
   public:
    virtual ~allocator_mechanism();
    virtual void* allocate(size_t n, size_t alignment,
                           void* hint = 0) = 0;
    virtual void deallocate(void* p, size_t n) = 0;

    virtual size_t max_size() const = 0;
  };
}
```

This abstract base class is the key to having runtime allocator selection. Defining a class derived from `allocator_mechanism` is also much easier than creating an allocator from scratch. There is no need to define a series of `typedefs` or the arcane `rebind` template. In fact, I have often seen people make the mistake of deriving an allocator from `std::allocator`, forgetting to define `rebind` and wondering why their allocator worked for `vector` but not for `list`.

20.6.c.1 `allocator_mechanism` members [allocator.mech.members]

```
virtual ~allocator_mechanism();
```

effects: Abtract-class destructor does nothing.

```
virtual void* allocate(size_t n, size_t alignment,
                       void* hint = 0) = 0;
```

Returns: A derived-class must override this function to return *n* bytes of memory with the specified *alignment* or else throw an appropriate exception. If *hint* is specified, a derived class may be used to optimize memory allocation (e.g., return a block as close as possible to *hint*).

Note: It is unspecified whether over-aligned requests are supported. A derived-class may honor the over-aligned request, silently ignore the alignment request, or throw an exception.

```
virtual void deallocate(void* p, size_t n) = 0;
```

Requires: p shall be a pointer obtained from `allocate()` and not yet deallocated; *n* shall be the value passed as the first argument to the invocation of `allocate()` that returned *p*.

Effects: A derived-class must override this function to the storage referenced by *p*.

```
virtual size_t max_size() const = 0;
```

Returns: A derived-class must override this function to return the largest number of bytes that can reasonably be returned from this object. The value returned from this function is not guaranteed to be available for allocation(). An implementation is permitted to assume that the value of `max_size()` does not change for the life of the object.

`allocator_mechanism` is a simple abstract base class for implementing polymorphic allocators. The requirements for `max_size()` do not require potentially expensive capacity computations.

20.6.b Class template `polymorphic_allocator` [polymorphic.allocator]

An instance of `polymorphic_allocator` is implicitly convertible from a pointer to `allocator_mechanism` and meets the requirements of an `Allocator` ([allocator.requirements] 20.1.2). Descriptions are provided here only for operations on polymorphic_allocator that are not described [allocator.requirements] or for operations where there is additional semantic information.

```
namespace std {
  // specialize for void:
  template <> class polymorphic_allocator<void> {
   public:
    typedef void* pointer;
    typedef const void* const_pointer;
    // reference-to-void members are impossible.
    typedef void value_type;
    template <class U> struct rebind {
      typedef polymorphic_allocator<U> other;
    };
  };

  template <class T> class polymorphic_allocator {
   public:
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;
    typedef T value_type;
    template <class U> struct rebind {
      typedef polymorphic_allocator<U> other;
    };

    polymorphic_allocator(allocator_mechanism *m = 0) throw();
    polymorphic_allocator(const polymorphic_allocator&) throw();
    template <class U>
      polymorphic_allocator(const polymorphic_allocator<U>&)
        throw();
    ~polymorphic_allocator() throw();

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

    pointer allocate(size_type,
      polymorphic_allocator<void>::const_pointer hint = 0);
    void deallocate(pointer p , size_type n );
    size_type max_size() const throw();

    void construct(pointer p, const T& val);
    template<class... Args>
      void construct(pointer p, Args&&... args);
    void destroy(pointer p);

    allocator_mechanism* mechanism();

   private:
    allocator_mechanism* mechanism_; // exposition only
  };
}
```

20.6.b.1 `polymorphic_allocator` constructors and destructor

```
polymorphic_allocator(allocator_mechanism *m = 0) throw();
```

Effects: if `m != 0`, `mechanism_ = m`, otherwise `mechanism_ = default_allocator_mechanism()`.

```
polymorphic_allocator(const polymorphic_allocator& a) throw();
template <class U>
  polymorphic_allocator(const polymorphic_allocator<U>& a)
    throw();
```

Effects: `mechanism_ = a.mechanism_`.

```
~polymorphic_allocator();
```

Effects: none

20.6.b.2 `polymorphic_allocator` members

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

Returns: the address of *x*, even in the presence of overloaded `operator&`.

```
pointer allocate(size_type n,
                 polymorphic_allocator<void>::const_pointer hint = 0);
```

Returns: `mechanism_->allocate(n*sizeof(T), alignof(T), hint);`

Throws: if `mechanism_->allocate()` throws an exception, then throw `bad_alloc`. [*Note:* does not rethrow the same exception as `mechanism_` unless the exception is also `bad_alloc` – *end note*]

This exception behavior produces expected results in code originally written with the default allocator in mind.

```
void deallocate(pointer p, size_type n);
```

Requires: p shall be a pointer obtained by calling `allocate()` on this same allocator instance (or an equal copy of the same type) and not yet deallocated; *n* shall be the value passed as the first argument to the invocation of `allocate()` that returned *p*.

Effects: `mechanism_->deallocate(p, n*sizeof(T));`

```
size_type max_size() const throw();
```

Returns: `mechanism_->max_size().`

```
allocator_mechanism* mechanism();
```

Returns: `mechanism_` (i.e., the pointer used to construct this object).

20.6.b.3 `polymorphic_allocator` type traits

```
namespace std {
  template <class T> struct
   suggest_scoped_allocator< polymorphic_allocator<T> >
     : true_type { };
```

20.6.b.4 `polymorphic_allocator` globals

```
template <class T, class U>
  bool operator==(const polymorphic_allocator<T>& x,
                  const polymorphic_allocator<U>& y) throw();
```

Returns: `x.mechanism() == y.mechanism().`

```
template <class T, class U>
  bool operator!=(const polymorphic_allocator<T>& x,
                  const polymorphic_allocator<U>& y) throw();
```

Returns: `x.mechanism() != y.mechanism()`.

20.6.c Class `new_delete_mechanism`

The `new_delete_mechanism` class is a concrete derived class of `allocator_mechanism` that implements storage allocation using `operator new()` and `operator delete()`.

```
namespace std {
  class new_delete_mechanism : public allocator_mechanism
  {
   public:
    virtual ~new_delete_mechanism();
    virtual void* allocate(size_t n, size_t alignment,
                           void* hint = 0);
    virtual void deallocate(void* p, size_t n);
    virtual size_t max_size() const;
  };
}
```

20.6.c.1 `new_delete_mechanism` members [allocator.mech.members]

```
virtual ~new_delete_mechanism();
```

effects: none.

```
virtual void* allocate(size_t n, size_t alignment,
                       void* hint = 0);
```

Returns: a pointer to the *n* bytes of storage with specified *alignment*, a. It is implementation-defined whether over-aligned requests are supported (3.11).

Remark: the storage is obtained by calling ::operator new(std::size_t) (18.5.1), but it is unspecified when or how often this function is called. The use of hint is unspecified, but intended as an aid to locality if an implementation so desires.

Throws: `bad_alloc` if the storage cannot be obtained.

```
virtual void deallocate(void* p, size_t n) = 0;
```

Requires: p shall be a pointer obtained from `allocate()` and not yet deallocated; *n* shall be the value passed as the first argument to the invocation of `allocate()` that returned *p*.

Effects: Deallocates the storage referenced by *p*.

`Remarks`: Uses `::operator delete(void*)` (18.5.1), but it is unspecified when this function is called.

```
virtual size_t max_size() const = 0;
```

Returns: the largest value *N* for which the call `allocate(`*N*`,1,0)` might succeed.

20.6.c Class template `allocator_mechanism_adapter`

The `allocator_mechanism_adapter` class adapts any `Allocator` class so that it can be used as a mechanism for constructing a `polymorphic_allocator`.

```
namespace std {
  template <class Allocator>
  class allocator_mechanism_adapter : public allocator_mechanism
  {
   public:
    typedef typename
      Allocator::template rebind<void>::other allocator_type;

    allocator_mechanism_adapter(
       const allocator_type& a = allocator_type());

    virtual ~allocator_mechanism_adapter();
    virtual void* allocate(size_t n, size_t alignment,
                           void* hint = 0);
    virtual void deallocate(void* p, size_t n);
    virtual size_t max_size() const;

   private:
    typename Allocator::template rebind<max_align_t>::other
      original_;   // exposition only
  };
}
```

20.6.c.1 `allocator_mechanism_adapter` members [allocator.mech.members]

```
allocator_mechanism_adapter(
    const allocator_type& a = allocator_type());
```

Effects: Constructs an `allocator_mechanism_adapter` with a copy of `a`.

```
virtual ~allocator_mechanism_adapter();
```

effects: `original_.~allocator_type()`.

```
virtual void* allocate(size_t n, size_t alignment,
                       void* hint = 0);
```

Returns: a pointer to the *n* bytes of storage with specified *alignment*, obtained by calling `allocate()` on the underlying allocator object. It is unspecified whether valid alignment requests less than the maximum fundamental alignment are rounded up to the maximum alignment.

```
virtual void deallocate(void* p, size_t n) = 0;
```

Requires: p shall be a pointer obtained from `allocate()` and not yet deallocated; *n* shall be the value passed as the first argument to the invocation of `allocate()` that returned *p*.

Effects: Deallocates the storage referenced by *p* by calling `deallocate()` on the underlying allocator.

```
virtual size_t max_size() const = 0;
```

Returns: the largest value *N* for which the call `allocate(`*N*`,1,0)` might succeed, obtained by calling `max_size()` on the underlying allocator.

20.6.d The default and global allocator mechanism

```
allocator_mechanism*
  set_default_allocator_mechanism(allocator_mechanism* m);
```

Effects: Sets the default allocator mechanism to be used when default-constructing a polymorphic allocator. If `m` is null, then sets the default allocator mechanism to a static object of type `new_delete_mechanism`. [*Note:* The intended purpose of setting the default allocator is to test that storage is being used correctly by intercepting unintended uses of the default store – *end note*]

```
allocator_mechanism*
  set_global_allocator_mechanism(allocator_mechanism* m);
```

Effects: Sets the global allocator mechanism for use in constructing static-duration objects (globals and singletons). If `m` is null, then sets the global allocator mechanism to a static object of type `new_delete_mechanism`. [*Note:* the use of the global allocator is voluntary, but recommended for static-duration objects that use allocators. The intended purpose is to allow testing that memory is being used correctly. – *end note*]

```
allocator_mechanism* default_allocator_mechanism();
```

Returns: The last value set using `set_default_allocator_mechanism()` or a pointer to a static object of type `new_delete_mechanism` if `set_default_allocator_mechanism()` was never called.

```
allocator_mechanism* global_allocator_mechanism();
```

Returns: The last value set using `set_global_allocator_mechanism()` or a pointer to a static object of type `new_delete_mechanism` if `set_global_allocator_mechanism()` was never called.

The intended purpose of these default mechanisms is to allow replacement of the default and global allocators for testing purposes. It is not recommended that they be used for any other purpose. These utilities must be standard, however, so that code can call them. In a testing environment, the default and global mechanisms would each be set to different test mechanisms, which would monitor intended and unintended uses of those allocators. The object under test should be constructed with a third test mechanism. If it is implemented correctly, then the default mechanism should be used only for transient allocations (e.g., local variables), whose lifetime is entirely within the scope of a single function. Separating the default from the global mechanism prevents intended long-term allocations from show up on the default mechanism’s count.

Question: What should the behavior of the default and global mechanism functions be in a threaded environment: thread-local? undefined? implementation-specified?

5. Polymorphic Allocator as the Default Allocator

Motivation

The most important quality of the polymorphic allocator is that it allows two objects of the same type to have different allocator mechanisms. The allocator is chosen at run time and is bound to an individual object rather than to its type. Ideally, one could pass a string using a custom allocator to any function that expects a string argument., even if that function were written years ago without consideration for allocators. Unfortunately, type `std::string` is currently defined to use the default allocator, not the polymorphic allocator. There is no way to pass a string using a polymorphic allocator to a function that expects an `std::string`. The same is true for `std::vector` and all of the other standard container classes.

Our solution is to make it so that any code compiled with C++0x will use the polymorphic allocator by default, but where the use of compile-time allocators is still honored.

Space and Time Considerations

When this idea was first proposed in Tremblant, Canada in 2005, there was concern that a polymorphic allocator is at least one pointer in size and that it would increase the footprint of string and every container type, even when the container is empty. A number of implementation tricks can avoid this overhead in the empty-container case:

- The allocator can be stored in the data field until the first allocation occurs, at which point it is copied into the allocated storage. A number of variations of this technique exist, with different tradeoffs (some tradeoffs varying with machine architecture).

- The previous technique can be enhanced further by "stealing" a bit from an existing field and using that bit to indicate whether or not the allocator uses the `new_delete_mechanism`. If it does use the `new_delete_mechanism`, the container does not need to store the allocator at all – the container can just use a global singleton allocator based on the `new_delete_alloactor`, with no additional space requirements in the typical case.

Our tests at Bloomberg indicate that the virtual function interface to the allocator mechanism adds almost nothing to the overall run time of an allocation-heavy function. In fact, the ability to replace the allocator at run-time has strong performance benefits in that it allows optimized allocators to be used in contexts where it was previously impossible. Moreover, our experience, since 2003, shows that third-party, allocator-oblivious code worked just fine with our modified standard library (built to support the scoped allocator model and a polymorphic default constructor) and was able to take advantage of our special allocators without source-code modifications.

Proposed Wording

Rename the "The default allocator" section to "The new-delete allocator".

Rename `allocator` to `new_delete_allocator`

Rename the "Class template `polymorphic_allocator`" section in the previous proposal to "The default allocator".

Rename `polymorphic_allocator` to `allocator`.

Need guidance: `new_delete_mechanism` can be implemented this way:

```
     typedef
       allocator_mechanism_adapter<new_delete_allocator<void> >
         new_delete_mechanism;
```

Should that definition be required, permitted, or disallowed?

6. Semantics of `swap`

Motivation

LWG 431 and N1599 point out that by the current definition of `swap` for containers, two containers of the same type can always be swapped in constant time and with no exception thrown. However, if the two containers contain stateful allocators and if those allocators do not compare equal, a question arises as to what `swap` should do. Should it swap the allocators, or should it do a linear-time swap of the contents of the containers?

The most recent proposed resolution to LWG 431 is to swap the allocators iff the allocator type itself is swapable. Implementing this semantic will require concepts. Never-the-less, it is the right thing to do *if* the containers use a certain allocator model. In proposal 3, above, I introduced the "Moves with Value" allocator model and the "Scoped" allocator model. In the former case, the allocator is copied when the container is copy-constructed. In the latter case it is not. Swapping the allocators is the *right* thing to do if the containers conform to the "Moves with Value" allocator model and absolutely the *wrong* thing to do if the containers conform to the "Scoped" allocator model. With the two allocator models well-defined, the desired behavior becomes clear.

Proposed Wording

TBD. Exact wording pending.

Rough wording, for each C::swap function:

> If `uses_scoped_allocator<C>::value` is false and `C::allocator_type` is `Swappable`, then swaps both the value *and* the allocator. Otherwise, swap only values. If the allocator is swapped, then the operation has constant complexity and does not throw unless the swap operation for the Compare, Hash, or Pred object throws. Otherwise, if the allocators compare equal (the typical case), the operation has constant complexity and does not throw (unless Compare, Hash, or Pred throw). Otherwise, the operation has linear complexity in the size of both containers combined and may throw an exception.

7. Semantics of `pointer` and `address` in Allocators

Motivation

The allocator requirements in [allocator.requirements] (20.1.2) gives the allocator author freedom to use a `pointer` type other than `value_type*` (i.e., a smart pointer type), in order to be able to allocate memory in unconventional ways. As described in **LWG 401** and **LWG 634**, the definition `pointer` and of the `address()` member function of allocators is incomplete. The intention of this section of this proposal is to clarify the language in the working paper such that, given a reference to an object allocated using an allocator, the `address()` function should recover the pointer returned by the `allocate()` function, even if `pointer` and `const_pointer` are other than `value_type*` and `const value_type*`, respectively.

Proposed Wording

In section [allocator.requirements] (20.1.2), table 40, modify the rows that describe `pointer` as follows:

`X::pointer`	Pointer to T	meets the requirements of a mutable random-access iterator ([random.access.iterators] 24.1.5); convertible to X::const_pointer.
`X::const_pointer`	Pointer to const T	meets the requirements of a random-access iterator ([random.access.iterators] 24.1.5)

The above changes define "Pointer to T." A random-access iterator is needed to ensure that, when allocating more than one object (e.g., in a vector), the resulting array can be indexed and that it is possible to determine whether a given pointer object points into the allocated range. (Note that *random-access iterator* will eventually be a concept.)

In section [allocator.requirements] (20.1.2), table 40, modify the rows that describe `address()` as follows:

`a.address(r)`	`X::pointer`	equivalent to p
`a.address(s)`	`X::const_pointer`	equivalent to q

We also clarify the notion that, after dereferencing a `pointer` object, you can reconstitute the original pointer (or something equivalent) by calling `address()` on the reference. (For pointers not allocated from `a1`, see additional proposed changes to `address()`, below.)

Change the rows that describe `construct` and `destroy` as follows. (Note that this is the first of two changes proposed in this document for this portion of table 40.):

`a.construct(p,t)`	(not used)	Effect: ~~::new((void*)p) T(t)~~ Constructs a copy of t at p. If t is an rvalue, it is forwarded to T's constructor as an rvalue, else it is forwarded as an lvalue.
`a.construct(p,v)`	(not used)	Effect: ~~::new((void*)p) T(std::forward<V>(v))~~ Constructs a T object from v at p. If v is an rvalue, it is forwarded to T's constructor as an rvalue, else it is forwarded as an lvalue.
`a.destroy(p)`	(not used)	Effect: ~~((T*)p)->~T()~~Destroys the object at p.

The above change addresses LWG 401 using the exact wording in the proposed resolution. The change ensures that `construct` and `destroy` do the "right thing" if `pointer` is not a true pointer.

One question remains: would a container that conforms to this clause really be able to manage memory through non-raw pointers? Does anybody have implementation experience with such odd-ball allocators?

In section [allocator.members] (20.6.1.1), modify the first two paragraphs as follows:

20.6.1.1 allocator members [allocator.members]

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

Returns: ~~&x~~ The actual address of x, even in the presence of an overloaded operator&.

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

Returns: ~~&x~~ The actual address of x, even in the presence of an overloaded operator&.

The above change addresses LWG 634 using the exact wording in the proposed resolution. It ensures that `std::allocator<T>::address()` does the right thing if `operator&` is overloaded for `T`. Note that this definition of `address` applies only to the default allocator (though it makes sense for any allocator for which `pointer` is the same as `value_type*`).

8. Allow the Use of `address` for Foreign Objects

Motivation

LGW 635 asserts that `address` should also work on objects *not* allocated by the allocator, as well as those that come from `allocate()`. This has some intuitive appeal and makes it easy to, e.g., check if an argument has an address that falls within the object being manipulated. I lean towards including this requirement for the reason that someone writing an allocator with a custom `pointer` type must be a brain surgeon whereas someone writing a container should not need to be as skilled.

Proposed Wording

In section [allocator.requirements] (20.1.2), table 39, modify the row that describes `t` and add a new row for `w` as follows:

`w`	a value of type `T&`
`t`	a value of type `const T&` obtained by conversion from a value w

In section [allocator.requirements] (20.1.2), table 40, add more rows to the description of `address()` as follows:

`a.address(w)`	`X::pointer`	`*a.address(w)` is identical to `w`
`a.address(t)`	`X::const_pointer`	`*a.address(t)` is identical to `t`

Guidance needed: This requirement may impose an unacceptable penalty for certain allocators. For example, assume a shared-memory pointer that contains a process-independent page ID for the shared memory and an offset into the shared memory. Only pointers to objects allocated from the allocator's shared memory space can be represented by such pointers. Requiring that every object have a corresponding pointer might require that the `pointer` type have an additional "raw" pointer to handle objects that don't come from the allocator. Without the universal address requirement, if a container really needs to compare addresses, it can dereference the pointer and use something like boost::addressof internally to get a raw address but this, as I said would require more skill from the container author.

9. Consistent Copy and Equality Semantics for Allocators

Motivation

[allocator.requirements] 20.1.2, table 40 requires that two allocators of the same type compare equal if memory allocated through one allocator can be deallocated through the other. It also states that if `X` and `Y` are corresponding allocators for different types, `T` and `U`, and if `a` is of type `X` and `b` is of type `Y`, then `X a(b)` will yield the post-condition that `Y(a) == b`. In other words, conversion is reversible. This comes close to, but does not fully state, that `operator==` for allocators must be transitive and reflexive, and that `Y(a) == Y(a)`.

As intuitive as these relationships may seem to some, there are reasoned opinions that these requirements are not needed and that there are useful allocators that could be built if these requirements were not present. For example, a small arena allocator that contains an array of bytes right within its footprint, would not even be equal to a copy of itself. Never the less, I propose that `operator==` be both transitive and reflexive and that copy-construction imply that the copy compares equal to the original. The reasons are as follows:

1. It violates a principle of operator overloading that an operator have semantics vastly different from the standard meaning. For example, `operator+` should not mean multiplication.
2. Similarly, it is not reasonable to assume that copy-constructing an object will yield an object that does not compare equal to the original.
3. Many containers have already been written that make the standard assumptions about copy construction and `operator==`.
4. Some uses of allocators, such as type erasure or footprint optimizations require that an allocator be able to allocate a copy of itself. At least one implementation of `vector` that I've seen puts the allocator on the heap.

A stateful allocator in this proposal would be required to share state with all of its copies (including copy-conversions). However, the benefits of having an allocator with truly unique state can be obtained by using an allocator with shared state and bundling the state object with the container that uses the allocator.

Proposed Wording

In section [allocator.requirements] (20.1.2), add the following to Table 40:

`a1 == a2`	`bool`	returns true iff storage allocated from each can be deallocated via the other. Equality is reflexive and transitive.
`a1 != a2`	`bool`	same as `!(a1 == a2)`
`X()`		creates a default instance [*Note:* destructor s assumed. – *end note*]
`X a1(a);`		post: `a1 == a`
`X a(b);`		post: `Y(a) == b` post: `a == X(b)`

These changes make copy-construction, comparison, and equality consistent with one another and with the common understanding of how they work.

10. Add variadic `construct` Requirement for Allocators

Motivation

The adoption of N2268 into the working paper introduced placement insert operations with variadic template arguments into containers. As per proposal 2 in this paper, containers should construct objects through the allocator's `construct()` function. This necessitates that `construct()` take variadic arguments.

Proposed Wording

In [allocator.requirements] (20.1.2), add the following row to Table 39:

`Args`	a template parameter pack
`args`	a function parameter pack with the pattern `Args&&`

Change the rows that describe `construct` and `destroy` as follows. (Note that this is the second of two changes proposed in this document for this portion of table 40. The text below shows the cumulative change.):

`a.construct(p,t)`	(not used)	Effect: ~~::new((void*)p) T(t)~~ Constructs a copy of t at p. If t is an rvalue, it is forwarded to T's constructor as an rvalue, else it is forwarded as an lvalue.
~~`a.construct(p,v)`~~	~~(not used)~~	~~Effect: ::new((void*)p) T(std::forward<V>(v)) Constructs a T object from v at p. If v is an rvalue, it is forwarded to T's constructor as an rvalue, else it is forwarded as an lvalue.~~
`a.construct(p,args)`	(not used)	Effect: Constructs a T object from args at p. args is passed to T's constructor as forward<Args>(args)...

11. Correction to `function` interface

TBD

Function objects need the `uses_scoped_allocator` trait and need to follow the scoped allocator rules.

12. Allocator-aware stringstream

TBD

`stringstream`, `istringstream`, and `ostringstream` allocate memory and should allow the user to control that allocation via an allocator argument at construction time.

13. Acknowledgements

I want to thank John Lakos for introducing me to the “Scoped” allocator model and showing me how powerful a consistent allocator model can be. I also want to thank John along with Vladimir Kliatchko and Shawn Edwards for their unwavering encouragement, support and confidence in me. Finally, thank you to Ion Gaztañaga for his input, and Howard Hinnant for introducing me to Ion.