Doc No: N2554=08-0064
Date: 2008-02-29
Author: Pablo Halpern
Bloomberg, L.P.

phalpern@halpernwightsoftware.com

The Scoped Allocator Model (Rev 2)

Contents

Changes from N2523 2
Changes from N2446 2
Motivation 2
Summary of Changes to the Working Draft 6
Usage Example 9
Document Conventions 11
Proposed Wording 11
  Allocator-related Type Traits 11
  Scoped Allocator Adaptor 16
  Element construction 22
  Pair changes 23
  Tuple Changes 25
  Container Requirements 28
  basic_string Changes 33
  deque changes 34
  list changes 34
  vector changes 35
  Changes to adapters 35
  map Changes 37
  multimap changes 37
  set changes 38
  multiset changes 38
  unordered_map changes 39
  unordered_multimap changes 39
  unordered_set changes 40
  unordered_multiset changes 40
  Function Changes 41
Implementation Experience 42

Changes from N2523

This document is a minor revision of N2523 incorporating typographical corrections only.

Changes from N2446

This revision of the Scoped Allocator Model proposal addresses the major concerns raised during the October 2007 meeting in Kona and adds one enhancement that was also requested during the Kona meeting. Specifically:

- Added alternative wording for concepts.
- Added diagrams to clarify the motivation section.
- A `scoped_allocator_adaptor` template has been added to allow one allocator to be specified for a container and a different allocator to be specified for its constituent elements.
- Scoped allocators are more flexible in this proposal: the inner allocator may be different from the outer allocator.
- The (conditional) change in container copy semantics (and its ripple effect on `swap()`) has been removed. A separate proposal (N2525) addresses the semantics of copy, move, and swap in the presence of unequal allocators. The two issues are no longer tied together.
- The constructor changes to `pair` and `tuple` are different in this proposal than in N2446 to correct for ambiguities with the new variadic constructors.
- A change to `function` was added to allow it to work seamlessly with the allocator framework.
- Minor changes to reflect additional implementation experience with new features.

Motivation

Memory is a core (pun intended) resource used by every part of a software system. Because memory organization and usage vary widely with the type of software being developed, every standard library component (and most library components outside of the standard) should provide a mechanism for the user to control how memory is allocated within that component. In the portion of the C++ standard library that evolved from the STL, user control over memory allocation is provided through the allocator parameter to the container templates.

The allocator instance passed to the constructor of a container is used by that container to allocate its internal data structures. Allocators thus provide control over the use of memory by a container. Containers are often nested, however; you can have a vector of strings or a map of lists of sets of user-defined types. In the C++03 allocator model (the *traditional allocator model*), the allocator instance for every element in a container may manage a different memory resource. For example, *Figure 1* illustrates a vector of strings using the traditional allocator model (rectangles represent data structures, ellipses represent memory resources managed by allocator instances):

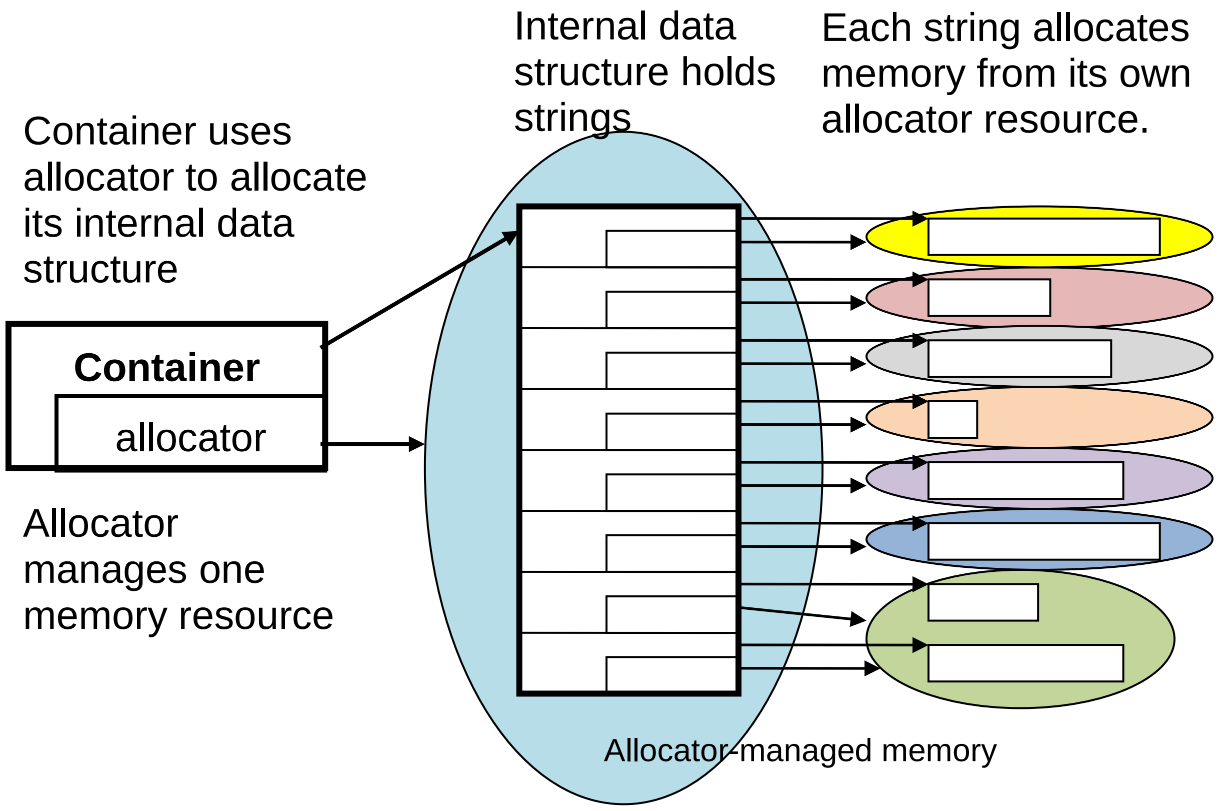
Internal data structure holds strings
Each string allocates memory from its own allocator resource.
Container uses allocator to allocate its internal data structure
Container
allocator
Allocator manages one memory resource
Allocator-managed memory

Figure 1: A container of strings using the traditional allocator model

The scoped allocator model provides a mechanism for the user to designate one allocator instance for use by the container (the *outer allocator*), and another (possibly different) allocator instance to be used by all of the container's elements (the *inner allocator*). The container ensures that all of its elements are constructed with a copy of the inner allocator instance. For example, we might want to place all strings into a special string region managed by a corresponding string allocator, as shown in *Figure 2*:

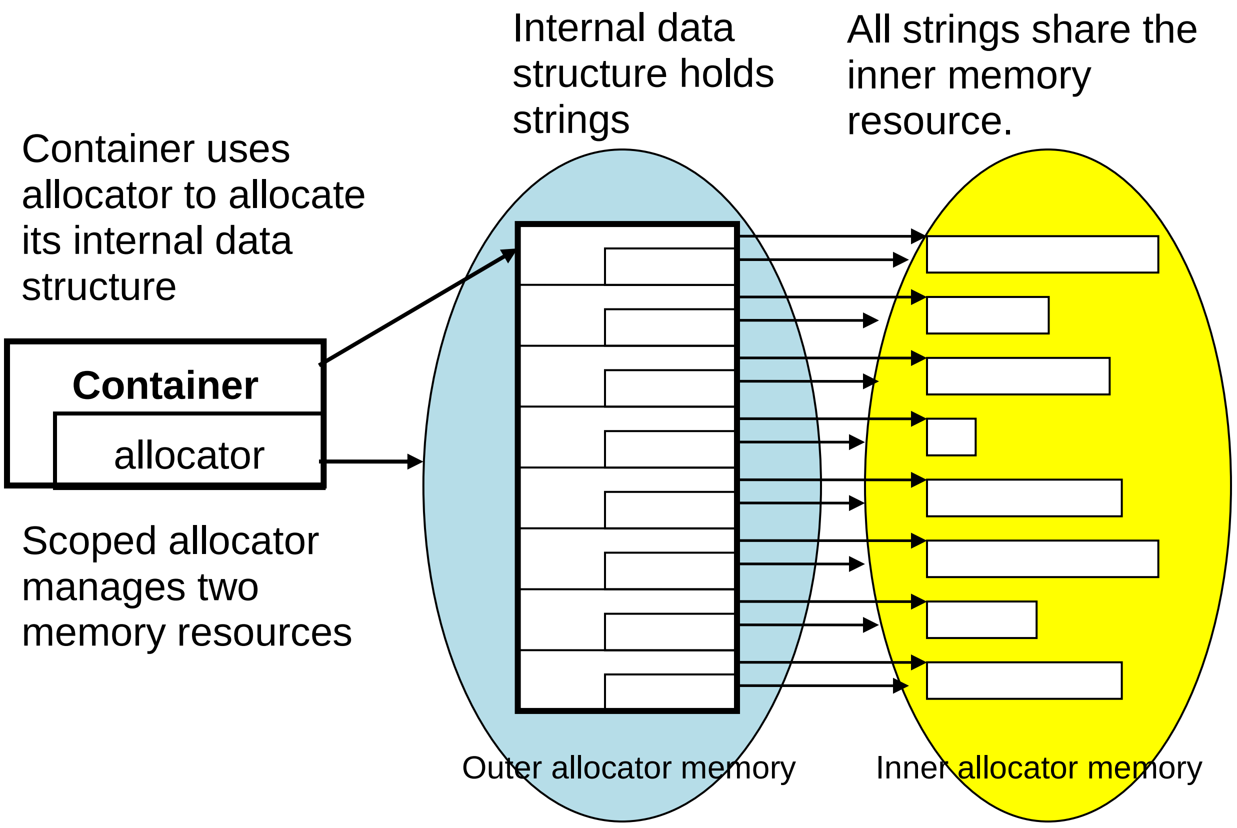
Internal data structure holds strings
All strings share the inner memory resource.
Container uses allocator to allocate its internal data structure
Container
allocator
Scoped allocator manages two memory resources
Outer allocator memory
Inner allocator memory

Figure 2: A container using a scoped allocator

In order to control the source of memory used by elements in a container, the traditional allocator model requires that the user control the allocator instance at the point where each element is inserted into the container. The purpose of the *scoped* allocator model, in contrast, is to allow a programmer to specify the memory resource to be used by every element of a container *at the point of construction of the container*, rather than at every insertion point.

The most common use of a scoped allocator is to ensure that all of the elements of a container get their memory from the same source as the container itself, i.e., the inner allocator is the same as the outer one. For example, if a container allocates memory from a local pool that maximizes locality-of-reference, it is desirable that all of its elements get their memory from the same local pool. Similarly, if a container allocates memory from a shared memory region (using fancy pointers), we would want all of its elements to allocate their memory from the same shared-memory region. In both of these examples, we would use a scoped allocator where the outer and inner allocators both allocate from the same memory resource, as shown in *Figure 3*:

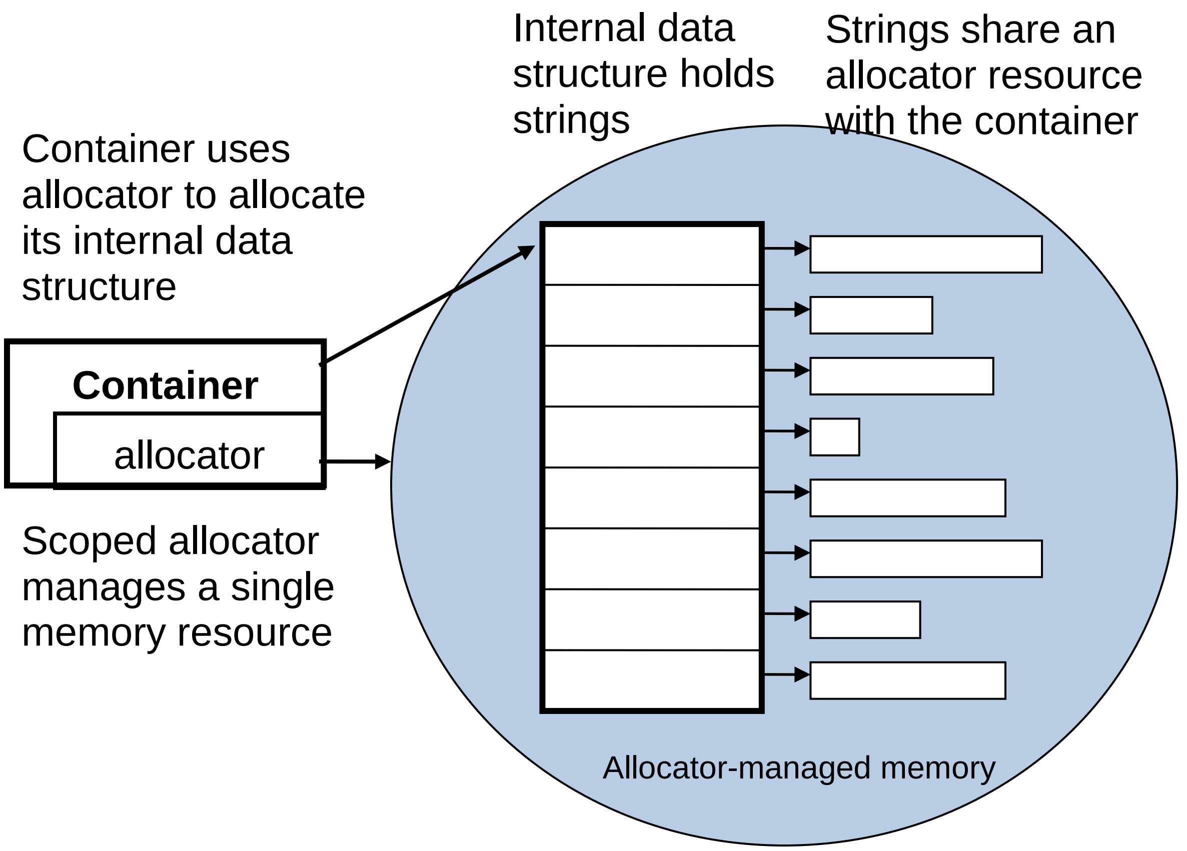
Internal data structure holds strings
Strings share an allocator resource with the container
Container uses allocator to allocate its internal data structure
Container
allocator
Scoped allocator manages a single memory resource
Allocator-managed memory

Figure 3: A container using a scoped allocator where the inner allocator is the same as the outer allocator

Note that in both of the scoped allocator examples, the invariant holds that `v[x].get_allocator() == v[y].get_allocator()` for all indexes *x* and *y* in vector *v*. This invariant makes it easier to reason about the behavior and performance of permuting algorithms like `sort()` and `remove_if()`.

The scoped allocator model was developed at multiple companies to address a real need for better control over how library classes use memory. In adapting the Bloomberg LP implementation experience for standardization, and in integrating it with other concerns regarding allocators, I was struck by how the C++ object model treats memory very differently from other system resources. Unlike the file-system, threads, concurrency locks, or other resources, access to memory is not filtered through a high-level abstraction: The language itself specifies certain rules for the relationship between elements in an array, the address of the first element of a structure, and the alignment of elements in a union. Although a certain amount of abstraction can be added through the use of smart pointers and classes that allocate memory (allocators), these abstractions are necessarily leaky; any attempt to completely abstract away access to memory is likely to result in substantial efficiency penalties.

The changes in this revised proposal make allocators as useful as possible to the largest audience possible, while remaining within the constraints of the proposed language and retaining backward-compatibility with C++03 allocator usage. With this proposal, we strive to provide good support for *both* the traditional allocator model and the scoped allocator model. Any added complexity, in our experience, is more than balanced by the added power for controlling memory allocation compared to the traditional allocator model alone.

Summary of Changes to the Working Draft

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

This summary assumes the presence of concepts, because concepts make it easier to express the intent. However, because concepts have not been accepted into the working paper as of this writing, the actual proposed wording is not concept-based, but does propose concept-based alternatives to most sections. In a sense, then, this proposal is really *two* proposals: the non-concept-based proposal intended to be replaced by the concept-based proposal when concepts are added to the language and library.

We begin with a new concept:

```
concept ScopedAllocator<class Alloc>
{
    requires Allocator<Alloc>;
    typename inner_allocator_type;
    inner_allocator_type Alloc::inner_allocator() const;
}
```

The author for a given allocator type, `Alloc`, should create a concept map for `ScopedAllocator<Alloc>` if `Alloc` is a “scoped” allocator, i.e., it is intended to be used according to the scoped allocator model. The traits-based alternative to this concept is:

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

Unless specialized for a given `Alloc` type, `is_scoped_allocator` inherits from `false_type`.

To make it easy to build scoped allocators from existing allocators, we add a new adaptor template:

```
template <Allocator Outer, Allocator Inner = NullAllocator>
class scoped_allocator_adaptor;
```

A `scoped_allocator_adaptor` is a scoped allocator with `Outer` as the outer allocator type and `Inner` as the inner allocator type. If instantiated with only one argument, then the outer and inner allocators are the same type and value. A scoped allocator adaptor can be nested to arbitrary depth by instantiating it with an inner allocator that is itself a scoped allocator adaptor.

Every constructor of every library class that uses an allocator must have a variant that takes an allocator argument, typically as an optional last constructor argument. The standard container classes, for example, are each enhanced with an *allocator-extended move constructor* and *allocator-extended copy constructor* as follows (where `C` represents the container class):

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

These allocator-extended constructors are used by the scoped allocator framework to pass the inner allocator from the container to each element as it is inserted.

Special constructors are also added to `pair`, `tuple`, and `function` so that they, too, can be constructed with allocators if they contain elements that use allocators. However, in order to avoid ambiguities with variadic constructors, these types use an allocator *prefix* argument (instead of a *suffix* argument) with a special type, `allocator_arg_t`, to clearly disambiguate the extended case from the non-allocator case. For example, the allocator-extended copy constructor for pair looks like this:

```
template <class Alloc>
pair(allocator_arg_t, const Alloc&, const pair&);
```

Two auto concepts automatically determine the order of arguments for its allocator-extended constructors of a given type:

```
auto concept
ConstructibleWithAllocatorSuffix<class T, class Alloc,
                                  class... Args>
{
    typename allocator_type;
    requires Allocator<allocator_type> &&
             Convertible<Alloc, allocator_type>;

    T::T(Args..., Alloc);
}
```

```
auto concept
ConstructibleWithAllocatorPrefix<class T, class Alloc,
                                  class... Args>
{
    typename allocator_type;
    requires Allocator<allocator_type> &&
             Convertible<Alloc, allocator_type>;

    T::T(allocator_arg_t, Alloc, Args...);
}
```

Or, using traits instead of concepts:

```
template <typename T>
struct constructible_with_allocator_suffix;

template <typename T>
struct constructible_with_allocator_prefix;
```

Unfortunately, these traits cannot be reliably deduced by library code without using concepts. Our real-world experience is in using explicit traits and we have found it to be workable. Still, it will be nice when it is no longer necessary to explicitly declare collaborative traits such as these.

Containers use the allocator-extended constructors to pass the inner allocator of a scoped allocator to each element as it is inserted. For each insertion function (including `insert`, `push_back`, `push_front`, and constructors that insert), the container constructs a new element using the `construct_element` function in the `ConstructibleAsElement` concept:

```
concept ConstructibleAsElement<class Alloc, class T,
                               class... Args>
{
    requires Allocator<Alloc>;
    void construct_element(Alloc& a, T* p, Args&& args...);
}
```

`ConstructibleAsElement` is a simple concept with a complex set of concept maps that automatically determine if `Alloc` is a scoped allocator and, if so, automatically passes `a.inner_allocator()` to `T`'s constructor using the appropriate allocator-extended constructor.

So that `queue`, `priority_queue`, `stack`, and `function` can be used as elements of containers with scoped allocators, allocator-extended constructors must be added to these templates. The `stringstream` class can also benefit from user-controlled allocation and thus we add an allocator argument to its constructor as well.

Usage Example

An arena is a mechanism that supplies memory from a contiguous buffer using very little bookkeeping. Typically, deallocating from an arena is a no-op, thus making an arena allocator inappropriate for applications where memory is allocated and then deallocated multiple times. However, the minimal bookkeeping and zero-cost deallocation make it ideal for situations where a data structure is built up over time, but destroyed all at once. All that is needed is to free the entire arena, possibly without even calling destructors. In the following example, we use a class, `SimpleArena` to supply memory from a fixed-sized buffer:

```
class SimpleArena
{
public:
    SimpleArena(std::max_align_t *buffer, std::size_t nbytes);

    void* alloc(std::size_t n);    // Allocate n bytes
    void dealloc(void *p);         // No-op
    std::size_t max_size() const; // return remaining bytes

private:
    std::max_align_t *buffer_;
    std::max_align_t *current_;  // Grows down toward start of buffer
};
```

We then build an allocator, `SimpleArenaAlloc`, that holds a pointer to a `SimpleArena` and uses it as a memory resource:

```
template <class TYPE>
class SimpleArenaAlloc
{
    SimpleArena *d_guts;

  public:
    typedef unsigned int   size_type;
    typedef int            difference_type;
    typedef TYPE*          pointer;
    typedef const TYPE*    const_pointer;
    typedef TYPE&          reference;
    typedef const TYPE&    const_reference;
    typedef TYPE           value_type;

    template<typename TYPE1>
    struct rebind
    {
        typedef SimpleArenaAlloc<TYPE1> other;
    };

    SimpleArenaAlloc(SimpleArena* g) throw() : d_guts(g) { }

    ...
};
```

Now, we build up a list of lists of `int`s. The advantage of an arena is compromised if parts of a data structure are not allocated from the same arena. If some parts of the structure are not allocated from the desired arena, then their memory will not be reclaimed when the arena is destroyed. Worse, if parts of the structure are allocated from a different arena and if that other arena is shorter-lived, then memory could be reclaimed to soon! We completely avoid these problems by creating a scoped allocator from our arena, thus ensuring that all parts of our list allocate memory from the same pool of memory:

```
    std::max_align_t buffer1[512/sizeof(std::max_align_t)];
    SimpleArena a1(buffer1, 512);

    typedef std::list<int, SimpleArenaAlloc<int> > InnerList;
    typedef std::scoped_allocator_adaptor<SimpleArenaAlloc<InnerList> >
        ScopedListAlloc;
    typedef std::list<InnerList, ScopedListAlloc> OuterList;

    // Allocate the outer list from the arena
    OuterList *bigList = new(a1.alloc(sizeof(OuterList)))
        OuterList(ScopedListAlloc(&a1));
```

We build up the list by constructing lists of ints and appending each to the list of lists:

```
    for (int i = 0; i < 3; ++i)
    {
        // Construct an inner list using a smaller arena.
        std::max_align_t buffer2[128/sizeof(std::max_align_t)];
        SimpleArena a2(buffer2, 128);
        InnerList littleList(i, i, SimpleArenaAlloc<int>(&a2));

        bigList->push_back(littleList);
        assert(bigList->back() == littleList);
```

Using the traditional allocator model, the `push_back` would insert an item using the same allocator as the argument (with dire results). However, using the scoped allocator model, the item is always constructed with the inner allocator, which in this case is the same as the outer allocator:

```
        ASSERT(bigList->back() == littleList);
        ASSERT(bigList->back().get_allocator() !=
               littleList.get_allocator());
        ASSERT(bigList->back().get_allocator() ==
               bigList->get_allocator());
    }
```

At this point, `bigList` points to a dynamically-allocated list of list of integers, all of the memory for which was allocated in the arena called a1. When `a1` goes out of scope, all of the memory used by `bigList` will be reclaimed, without ever calling a list destructor!

Document Conventions

All section names and numbers are relative to the October 2007 working draft, N2461.

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).

As of this writing, concepts have not yet been accepted into the working draft. Accordingly, the proposed wording does not use concepts. Alternative working-paper text that declares concepts and concept maps is enclosed in a red box. Even once concepts are accepted, it is hoped that the non-concept interface could define a de-facto standard method of implementing the elements of this proposal using a C++03 compiler.

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

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

Proposed Wording

Allocator-related Type Traits

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

```
// 2.6.w, allocator-argument tag
struct allocator_arg_t { };
const allocator_arg_t allocator_arg = allocator_arg_t();
```

A `struct` rather than an `enum` was chosen because the implicit enum-to-int conversion could potentially re-introduce the ambiguity that this type is intended to avoid.

Insert after the preceding (or else in <memory_concepts>):

```
// 2.6.x, allocator-related traits
template <class T, class Alloc> struct uses_allocator;
template <class Alloc> struct is_scoped_allocator;
template <class T> struct constructible_with_allocator_suffix;
template <class T> struct constructible_with_allocator_prefix;
```

// 2.6.x, allocator-related concepts

```
concept ScopedAllocator<typename Alloc>  see below
auto concept UsesAllocator<class T, class Alloc>  see below
auto concept
ConstructibleWithAllocatorSuffix<class T, class Alloc,
                                  class... Args>  see below
auto concept
ConstructibleWithAllocatorPreffix<class T, class Alloc,
                                  class... Args>  see below
```

Insert before [default.allocator] (20.6.1):

2.6.w Allocator-argument tag

```
namespace std {
    struct allocator_arg_t { };
    const allocator_arg_t allocator_arg = allocator_arg_t();
}
```

The `allocator_arg_t` struct is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, several types (see `pair` (20.2.3)) have constructors with `allocator_arg_t` as the first argument, immediately followed by an argument of type that satisfies the `Allocator` requirements (20.1.2).

Insert after the preceding (or else in <memory_concepts>):

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

`template <class T, class Alloc> struct uses_allocator;` *see below*

Remark: Automatically detects if `T` has a nested `allocator_type` that is convertible from `Alloc`. Meets the *BinaryTypeTrait* requirements ([meta.rqmts] 20.4.1). A program may specialize this type to derive from `true_type` for a `T` of user-defined type if `T` does not have a nested `allocator_type` but is nonetheless constructible using the specified `Alloc`.

Result: derived from `true_type` if `Convertible<Alloc,T::allocator_type>` and derived from `false_type` otherwise.

The class templates, `is_scoped_allocator`, `constructible_with_allocator_suffix`, and `constructible_with_allocator_prefix` meet the *UnaryTypeTrait* requirements ([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 provided that the user-defined type meets the requirement of the trait. However, a program is never *required* to specialize these traits.

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

Remark: if a specialization is derived from `true_type`, indicates that `Alloc` is a *Scoped Allocator*. A scoped allocator specifies the memory resource to be used by a container (as any other allocator does) and also specifies an *inner allocator* resource to be used by every element in the container.

Requires: if a specialization is derived from `true_type`, `Alloc` is required to have an `inner_allocator_type` nested type and a member function `inner_allocator()`, which is callable with no arguments and which returns a type convertible to `inner_allocator_type`.

```
template <class T> struct constructible_with_allocator_suffix
    : false_type { };
```

Remark: if a specialization is derived from `true_type`, indicates that `T` may be constructed with an allocator as its last constructor argument. Ideally, all constructors of `T` (including the copy and move constructors) should have a variant that accepts a final argument of `allocator_type`.

Requires: if a specialization is derived from `true_type`, `T` must have a nested type, `allocator_type` and at least one constructor for which `allocator_type` is the last parameter. If not all constructors of `T` can be called with a final `allocator_type` argument, and if `T` is used in a context where a container must call such a constructor, then the program is ill-formed.

[*Example:*

```
template <class T, class A = allocator<T> >
class Z {
  public:
    typedef A allocator_type;

    // Default constructor with optional allocator suffix
    Z(const allocator_type& a = allocator_type());

    // Copy constructor and allocator-extended copy constructor
    Z(const Z& zz);
    Z(const Z& zz, const allocator_type& a);
};

// Specialize trait for class template Z
template <class T, class A = allocator<T> >
struct constructible_with_allocator_suffix<Z<T,A> >
    : true_type { };
```

-- *end example*]

```
template <class T> struct constructible_with_allocator_prefix
    : false_type { };
```

Remark: if a specialization is derived from `true_type`, indicates that `T` may be constructed with `allocator_arg` and `T::allocator_type` as its first two constructor arguments. Ideally, all constructors of `T` (including the copy and move constructors) should have a variant that accepts these two initial arguments.

Requires: if a specialization is derived from `true_type`, `T` must have a nested type, `allocator_type` and at least one constructor for which `allocator_arg_t` is the first parameter and `allocator_type` is the second parameter. If not all constructors of `T` can be called with these initial arguments, and if `T` is used in a context where a container must call such a constructor, then the program is ill-formed.

[*Example:*

```
template <class T, class A = allocator<T> >
class Y {
  public:
    typedef A allocator_type;

    // Default constructor with and allocator-extended default constructor
    Y();
    Y(allocator_arg_t, const allocator_type& a);

    // Copy constructor and allocator-extended copy constructor
    Y(const Y& yy);
    Y(allocator_arg_t, const allocator_type& a, const Y& yy);

    // Variadic constructor and allocator-extended variadic constructor
    template<class ...Args> Y(Args&& args...);
    template<class ...Args>
    Y(allocator_arg_t, const allocator_type& a,
      Args&&... args);
};

// Specialize trait for class template Y
template <class T, class A = allocator<T> >
struct constructible_with_allocator_prefix<Y<T,A> >
    : true_type { };
```

-- *end example*]

The `constructible_with_allocator_suffix/prefix` traits are needed only because there is no way, without concepts or compiler support, to detect the constructor signatures for a given type.

2.6.x, allocator-related concepts

```
auto concept UsesAllocator<typename T, typename Alloc> {
    requires Allocator<Alloc>;
    typename T::allocator_type;
    requires Convertible<Alloc, allocator_type>;
}
```

Remark: Automatically detects if `T` has a nested `allocator_type` that is convertible from `Alloc`. A program may create a concept map for `UsesAllocator` for a `T` of user-defined type if `T` does not have a nested `allocator_type` but is nonetheless constructible using the specified `Alloc`.

```
concept ScopedAllocator<typename Alloc> {
    requires Allocator<Alloc>;
    typename inner_allocator_type;
    inner_allocator_type Alloc::inner_allocator() const;
}
```

Remark: a concept map for a give `Alloc` type indicates that `Alloc` is a *Scoped Allocator*. A scoped allocator specifies the memory resource to be used by a container (as any other allocator) and also specifies an *inner allocator* resource to be used by every element in the container.

```
auto concept
ConstructibleWithAllocatorSuffix<class T, class Alloc,
                                  class... Args>
    : UsesAllocator<T, Alloc>
{
    requires Constructible<T, Args..., allocator_type>;
}
```

Remark: an (automatically generated) concept map for a given set of parameters indicates that `T` may be constructed with `allocator_type` as its last constructor argument. Ideally, all constructors of `T` (including the copy and move constructors) should have a variant that accepts a final argument of `allocator_type`.

[*Example:*

```
  template <class T, class A = allocator<T> >
  class Z {
    public:
      typedef A allocator_type;

      // Default constructor with optional allocator suffix
      Z(const allocator_type& a = allocator_type());

      // Copy constructor and allocator-extended copy constructor
      Z(const Z& zz);
      Z(const Z& zz, const allocator_type& a);
  };
```

-- *end example*]

```
auto concept
ConstructibleWithAllocatorPrefix<class T, class Alloc,
                                  class... Args>
```

```
    : UsesAllocator<T, Alloc>
{
    requires Allocator<Alloc>;
    typename T::allocator_type;
    requires Convertible<Alloc, allocator_type>;
    requires Constructible<T, allocator_arg_t, Alloc, Args...>;
}
```

Remark: an (automatically generated) concept map for a given set of parameters indicates that `T` may be constructed with `allocator_arg` and `T::allocator_type` as its first two constructor arguments. Ideally, all constructors of `T` (including the copy and move constructors) should have a variant that accepts these two initial arguments.

[*Example:*

```
  template <class T, class A = allocator<T> >
  class Y {
    public:
      typedef A allocator_type;

      // Default constructor and allocator-extended default constructor
      Y();
      Y(allocator_arg_t, const allocator_type& a);

      // Copy constructor and allocator-extended copy constructor
      Y(const Y& yy);
      Y(allocator_arg_t, const allocator_type& a, const Y& yy);

      // Variadic constructor and allocator-extended variadic constructor
      template<class ...Args> Y(Args&&... args);
      template<class ...Args>
      Y(allocator_arg_t, const allocator_type& a,
        Args&&... args);
  };
```

-- *end example*]

The concepts for `ConstructibleWithAllocatorSuffix` and `ConstructibleWithAllocatorPrefix` will match only types that have an `allocator_type` associated type. The reason for this requirement is to avoid overly-general matches with template constructors – i.e., a constructor that takes a template argument of any type will take an allocator argument even if that type doesn't directly use allocators.

Scoped Allocator Adaptor

In section 20.6, before the declaration of `raw_storage_iterator`, insert:

// 20.6.1+, scoped allocator adaptor:

```
template<class OuterA, class InnerA = void>
  class scoped_allocator_adaptor;

template<class Alloc>
  class scoped_allocator_adaptor<Alloc, void>;

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

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

template<class OuterA, class InnerA>
  concept_map ScopedAllocator<
     scoped_allocator_adaptor<OuterA,InnerA> > { }

template<class OuterA, class InnerA>
  concept_map AllocatorPropagateNever<
     scoped_allocator_adaptor<OuterA, InnerA> > { }
```

Note that the declaration of the trait `allocator_propagate_never` (or the concept map for `AllocatorPropagateNever`) is contingent on acceptance of N2525.

```
template<typename OuterA1, typename OuterA2, typename InnerA>
  bool operator==(const scoped_allocator_adaptor<OuterA1,InnerA>& a,
                  const scoped_allocator_adaptor<OuterA2,InnerA>& b);

template<typename OuterA1, typename OuterA2, typename InnerA>
  bool operator!=(const scoped_allocator_adaptor<OuterA1,InnerA>& a,
                  const scoped_allocator_adaptor<OuterA2,InnerA>& b);
```

Between sections 20.6.1 and 20.6.2, add a new subsection:

20.6.x Scoped Allocator Adaptor [scoped.allocator]

The `scoped_allocator_adaptor` class template is an allocator template that specifies the memory resource (the *outer allocator*) to be used by a container (as any other allocator does) and also specifies an *inner allocator* resource to be used by every element in the container. This adaptor is instantiated with outer and inner allocator types. If instantiated with only one allocator type (i.e., the second type is `void`), the same allocator type is used for both the outer and inner allocator types and the same allocator instance is used for both the outer and inner allocator instances. The interface is specialized for the single-allocator case such that it takes only one allocator instance argument in the constructor, verses two allocators for the general case. Otherwise, the interface to the specialized and general cases are the same. A `scoped_allocator_adaptor` that is instantiated with two identical parameters is different than an adaptor instantiated with only one parameter: the former may be constructed with different instances of outer and inner allocators whereas the second may be constructed only with one allocator instance. [*Note:* the `scoped_allocator_adaptor` is derived from the outer allocator type, so it can be substituted for the outer allocator type in most expressions. – *end note*]

```
namespace std {

  template<typename OuterA, typename InnerA = void>
    class scoped_allocator_adaptor ;

  template<typename OuterA>
    class scoped_allocator_adaptor<OuterA, void> : public OuterA
  {
  public:
    typedef OuterA outer_allocator_type;
    typedef OuterA inner_allocator_type;
        // outer and inner allocator types are the same.

    typedef typename outer_allocator_type::size_type       size_type;
    typedef typename outer_allocator_type::difference_type difference_type;
    typedef typename outer_allocator_type::pointer         pointer;
    typedef typename outer_allocator_type::const_pointer   const_pointer;
    typedef typename outer_allocator_type::reference       reference;
    typedef typename outer_allocator_type::const_reference const_reference;
    typedef typename outer_allocator_type::value_type      value_type;

    template <typename _Tp>
    struct rebind
    {
        typedef scoped_allocator_adaptor<OuterA::template rebind<_Tp>::other,
                                         void> other;
    };

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

    template <typename OuterA2>
      scoped_allocator_adaptor(
          scoped_allocator_adaptor<OuterA2, void>&&);
    template <typename OuterA2>
      scoped_allocator_adaptor(
          const scoped_allocator_adaptor<OuterA2, void>&);

   ~scoped_allocator_adaptor();

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

    pointer allocate(size_type n);
    template <typename _HintP>
      pointer allocate(size_type n, _HintP u);
    void deallocate(pointer p, size_type n);
    size_type max_size() const;
    template <class... Args>
      void construct(pointer p, Args&&... args);
    void destroy(pointer p);

    const outer_allocator_type& outer_allocator();
    const inner_allocator_type& inner_allocator();
  };

  template<typename OuterA, typename InnerA>
    class scoped_allocator_adaptor : public OuterA
  {
  public:
    typedef OuterA outer_allocator_type;
    typedef InnerA inner_allocator_type;

    typedef typename outer_allocator_type::size_type       size_type;
    typedef typename outer_allocator_type::difference_type difference_type;
    typedef typename outer_allocator_type::pointer         pointer;
    typedef typename outer_allocator_type::const_pointer   const_pointer;
    typedef typename outer_allocator_type::reference       reference;
    typedef typename outer_allocator_type::const_reference const_reference;
    typedef typename outer_allocator_type::value_type      value_type;

    template <typename _Tp>
    struct rebind
    {
        typedef scoped_allocator_adaptor<OuterA::template rebind<_Tp>::other,
                                         InnerA> other;
    };

    scoped_allocator_adaptor();
    scoped_allocator_adaptor(outer_allocator_type&& outerAlloc,
                             inner_allocator_type&& innerAlloc);
    scoped_allocator_adaptor(const outer_allocator_type& outerAlloc,
                            const inner_allocator_type& innerAlloc);
    scoped_allocator_adaptor(scoped_allocator_adaptor&& other);
    scoped_allocator_adaptor(const scoped_allocator_adaptor& other);

    template <typename OuterAlloc2>
      scoped_allocator_adaptor(
        scoped_allocator_adaptor<OuterAlloc2&,InnerA>&&);
    template <typename OuterAlloc2>
      scoped_allocator_adaptor(
        const scoped_allocator_adaptor<OuterAlloc2&,InnerA>&);

   ~scoped_allocator_adaptor();

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

    pointer allocate(size_type n);
    template <typename _HintP>
      pointer allocate(size_type n, _HintP u);
    void deallocate(pointer p, size_type n);
    size_type max_size() const;

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

    const outer_allocator_type& outer_allocator() const;
    const inner_allocator_type& inner_allocator() const;
  };

  template<typename OuterA1, typename OuterA2, typename InnerA>
  bool operator==(const scoped_allocator_adaptor<OuterA1,InnerA>& a,
                  const scoped_allocator_adaptor<OuterA2,InnerA>& b);

  template<typename OuterA1, typename OuterA2, typename InnerA>
  bool operator!=(const scoped_allocator_adaptor<OuterA1,InnerA>& a,
                  const scoped_allocator_adaptor<OuterA2,InnerA>& b);

}
```

20.6.x.1 scoped_allocator_adaptor constructors [scoped.adaptor.cntr]

```
scoped_allocator_adaptor();
```

effects: initializes the outer and inner allocator instances using their corresponding default constructors.

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

effects: initializes the outer and inner allocator instances from the corresponding parts of `other`.

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

requires: `scoped_allocator_adaptor` was instantiated with only one parameter.

effects: initializes the base class (which is both the outer and inner allocator) from `outerAlloc`.

```
template <typename OuterA2>
  scoped_allocator_adaptor(
       scoped_allocator_adaptor<OuterA2, InnerA>&& other);
template <typename OuterA2>
  scoped_allocator_adaptor(
       const scoped_allocator_adaptor<OuterA2, InnerA>& other);
```

requires: `Same<OuterA2, OuterA::rebind<value_type>::other>`.

effects: initializes the outer and inner allocator instances from the corresponding parts of `other`.

20.6.x.2 scoped_allocator_adaptor members [scoped.adaptor.members]

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

returns: `outer_allocator().address(x);`

```
pointer allocate(size_type n);
```

returns: `outer_allocator().allocate(n);`

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

returns: `outer_allocator().allocate(n, u);`

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

effects: `outer_allocator().deallocate(p, n);`

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

returns: `outer_allocator().max_size();`

```
template <class... Args>
  void construct(pointer p, Args&&... args);
```

effects: `outer_allocator().construct(p, args...);`

```
void destroy(pointer p);
```

effects: `outer_allocator().destroy(p);`

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

returns: the outer allocator used to construct this object.

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

returns: the inner allocator used to construct this object. For the single-parameter instantiation, returns the same reference as `outer_allocator()`.

20.6.x.3 scoped_allocator_adaptor globals [scoped.adaptor.globals]

```
template<typename OuterA1, typename OuterA2, typename InnerA>
bool operator==(const scoped_allocator_adaptor<OuterA1, InnerA>& a,
                const scoped_allocator_adaptor<OuterA2, InnerA>& b);
```

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

```
template<typename OuterA1, typename OuterA2, typename InnerA>
bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerA>& a,
                const scoped_allocator_adaptor<OuterA2, InnerA>& b);
```

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

Element construction

In section 20.6 [memory], after the comment that reads "//specialized algorithms", insert the following:

```
template<typename Alloc, typename T, class... Args>
  void construct_element(Alloc& alloc, T& r, Args&&... args);
```

Before section 20.6.4.1 [uninitialized.copy], insert the following:

20.6.4.x `construct_element` **[construct.element]**

```
template<typename Alloc, typename T, class... Args>
  void construct_element(Alloc& alloc, T& r, Args&&... args);
```

remarks: This function is called from within containers in order to construct elements during insertion operations as well as to move elements during reallocation operations. It automates the process of determining if the scoped allocator model is in use and transmitting the inner allocator for scoped allocators.

effects:

```
If ScopedAllocator<Alloc> and UsesAllocator<T, A::inner_allocator_type>
then
  If ConstructibleWithAllocatorPrefix<T,A::inner_allocator_type,Args…>
  then
    alloc.construct(alloc.address(r), allocator_arg_t,
                    alloc.inner_allocator(), args...);
  else if
   ConstructibleWithAllocatorSuffix<T,A::inner_allocator_type,Args…>
  then
    alloc.construct(alloc.address(r),args...,alloc.inner_allocator());
  else
    program is ill-formed
  end if
else
  alloc.construct(alloc.address(r), args...);
end if
```

This function encapsulates all of the mechanism (meta-programming and/or concept overloading) necessary to construct a container element. If a type has an `allocator_type` but does not indicate how to construct an item using an allocator (by specializing a trait or mapping a concept), then the program is ill-formed. Doing otherwise would violate the principle of least surprise in that a user could reasonably expect that the allocator would be transmitted to the element in that case. Note that the element argument to `construct_element` is of type `T&` rather than

`Alloc::reference` for two reasons: First, we want the call to be element-centric, rather than allocator-centric. Second, we are leaving the door open to a more permissive definition of `Allocator::construct` that would allow an allocator to construct an object other than one of type `Allocator::value_type`.

Pair changes

Modify the declaration of pair<T1, T2>, in section 20.2.3 [pairs] 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 V > pair(pair<U , V >&& p );
    template<class U, class... Args> pair(U&& x, Args&&... args);

    // allocator-extended constructors
    template <class Alloc> pair(allocator_arg_t, const Alloc& a);
    template <class Alloc>
      pair(allocator_arg_t,const Alloc& a,const T1& x,const T2& y);
    template<class U, class V, class Alloc >
      pair(allocator_arg_t, const Alloc& a, U&& x , V&& y);
    template <class Alloc>
      pair(allocator_arg_t, const Alloc& a, pair&& p);
    template<class U, class V, class Alloc>
      pair(allocator_arg_t, const Alloc& a, const pair<U, V >& p);
    template<class U , class V, class Alloc >
      pair(allocator_arg_t, const Alloc& a, pair<U, V>&& p);
    template<class U, class... Args>
      pair(allocator_arg_t, const Alloc& a, U&& x, Args&&... args);

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

    void swap(pair&& p );
};

template <class T1, class T2, class Alloc>
  struct uses_allocator<pair<T1, T2>, Alloc>;

template <class T1, class T2>
  struct constructible_with_allocator_prefix<pair<T1, T2> >;
```

Before the definition of the first constructor, insert these traits/concepts:

```
template <class T1, class T2, class Alloc>
struct uses_allocator<pair<T1, T2>, Alloc>
    : true_type { };
```

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

remarks: Specialization of this trait informs other library components that `pair` can be constructed with an allocator, even though it does not have an `allocator_type` associated type.

```
template <class T1, class T2>
struct constructible_with_allocator_prefix<pair<T1, T2> >
    : true_type { };
```

remarks: Specialization of this trait informs other library components that a `pair` can always be constructed with an allocator prefix argument.

```
template <class T1, class T2, class Alloc>
concept_map UsesAllocator<pair<T1, T2>, Alloc> {
    typedef Alloc allocator_type;
}

template <class T1, class T2, class ...Args>
concept_map constructible_with_allocator_prefix<pair<T1, T2>,
                                                Args... > {
}
```

After the definition of all of the existing constructors, insert the following definitions:

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

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

effects: equivalent to the preceding constructors except that the allocator argument is passed conditionally to the constructors of `first`, `second`, or both. If `uses_allocator<T1, Alloc>::value && constructible_with_allocator_prefix<T1>::value`, then `first` is constructed with `allocator_arg`, and *a* as the first two constructor arguments.

Otherwise, if `uses_allocator<T1, Alloc>::value && constructible_with_allocator_suffix<T1>::value`, then `first` is constructed with *a* as the last constructor argument. Otherwise `first` is constructed without an allocator as in the preceding constructors. If `uses_allocator<T2, Alloc>::value && constructible_with_allocator_prefix<T2>::value`, then `second` is constructed with `allocator_arg`, and *a* as the first two constructor arguments. Otherwise, if `uses_allocator<T2, Alloc>::value && constructible_with_allocator_suffix<T2>::value`, then `second` is constructed with *a* as the last constructor argument. Otherwise `second` is constructed without an allocator, as in the preceding constructors. [*Note:* If both `first` and `second` are constructed without allocators, then the *a* argument is ignored – *end note*].

These definitions allow containers (especially associative containers) to pass an allocator to items of `pair` type. These constructors can be implemented using constructor delegation and meta-programming. In the existing implementations, these constructors are implemented by using inheritance and meta-programming.

Tuple Changes

In section 20.3.1 [tuple.tuple], modify the declaration of class template tuple as follows:

```
template <class... Types>
class tuple
{
public:
  tuple();
  explicit tuple(const Types&...);
  template <class... UTypes>
    explicit tuple(UTypes&&...);

  tuple(const tuple&);
  tuple(tuple&&);

  template <class... UTypes>
    tuple(const tuple<UTypes...>&);
  template <class... UTypes>
    tuple(tuple<UTypes...>&&);

  template <class U1, class U2>
    tuple(const pair<U1, U2>&); // iff sizeof...(Types) == 2
  template <class U1, class U2>
    tuple(pair<U1, U2>&&);      // iff sizeof...(Types) == 2

  template <class Alloc>
    tuple(allocator_arg_t, const Alloc& a);
  template <class Alloc>
    tuple(allocator_arg_t, const Alloc& a, const Types&...);
  template <class Alloc, class... UTypes>
    tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
  template <class Alloc>
    tuple(allocator_arg_t, const Alloc& a, const tuple&);
  template <class Alloc>
    tuple(allocator_arg_t, const Alloc& a, tuple&&);

  template <class Alloc, class... UTypes>
    tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
  template <class Alloc, class... UTypes>
    tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);

  template <class Alloc, class U1, class U2>
    tuple(allocator_arg_t, const Alloc& a,
          const pair<U1, U2>&);
  template <class Alloc, class U1, class U2>
    tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);

  tuple& operator=(const tuple&);
  tuple& operator=(tuple&&);

  template <class... UTypes>
    tuple& operator=(const tuple<UTypes...>&);
  template <class... UTypes>
    tuple& operator=(tuple<UTypes...>&&);

  template <class U1, class U2>
    tuple& operator=(const pair<U1, U2>&); // iff sizeof...(Types) == 2
  template <class U1, class U2>
    tuple& operator=(pair<U1, U2>&&); // iff sizeof...(Types) == 2
};
```

Before section 20.3.1.1 [tuple.cnstr], insert the following new subsection:

20.3.1.1 Tuple traits [tuple.traits]

```
template <class... Types, class Alloc>
struct uses_allocator<tuple<Types...>, Alloc>
    : true_type { };
```

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

remarks: Specialization of this trait informs other library components that `tuple` can be constructed with an allocator, even though it does not have an `allocator_type` associated type.

```
template <class... Types>
struct constructible_with_allocator_prefix<tuple<Types...> >
    : true_type { };
```

remarks: Specialization of this trait informs other library components that a `tuple` can always be constructed with an allocator prefix argument.

```
template <class... Types, class Alloc>
concept_map UsesAllocator<tuple<Types...>, Alloc> {
    typedef Alloc allocator_type;
}

template <class... Types, class ...Args>
concept_map constructible_with_allocator_prefix<tuple<Types...>,
                                                Args... > {
}
```

In section 20.3.1.1 [tuple.cnstr], after the definition of all of the existing constructors, add the following definitions:

```
template <class Alloc>
  tuple(allocator_arg_t, const Alloc& a);
template <class Alloc>
  tuple(allocator_arg_t, const Alloc& a, const Types&...);
template <class Alloc, class... UTypes>
  tuple(allocator_arg_t, const Alloc& a, UTypes&&...);

template <class Alloc>
  tuple(allocator_arg_t, const Alloc& a, const tuple&);
template <class Alloc>
  tuple(allocator_arg_t, const Alloc& a, tuple&&);

template <class Alloc, class... UTypes>
  tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
template <class Alloc, class... UTypes>
  tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);

template <class Alloc, class U1, class U2>
  tuple(allocator_arg_t, const Alloc& a,
   const pair<U1, U2>&);
template <class Alloc, class U1, class U2>
  tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);
```

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

effects: equivalent to the preceding constructors except that the allocator argument is passed conditionally to the constructors of each element. For each element *n* of type `T`*n*`, if uses_allocator<T`*n*`, Alloc>::value && constructible_with_allocator_prefix<T`*n*`>::value`, then the element *n* is constructed with `allocator_arg`, and ***a*** as the first two constructor arguments. Otherwise, if `uses_allocator<T`*n*`, Alloc>::value && constructible_with_allocator_suffix<T`*n*`>::value`, then element *n* is constructed with ***a*** as the last constructor argument. Otherwise element *n* is constructed without an allocator, as in the preceding constructors. [*Note:* If none of the elements are constructed without allocators, then the ***a*** argument is ignored – *end note*].

Container Requirements

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.~~
>
> Objects stored in these components shall be constructed using the `construct_element` specialized algorithm (20.6.4.x [construct.element]). For each operation that inserts an element of type `T` into a container (`insert`, `push_back`, `push_front`, `emplace`, etc.), with arguments `args`..., `T` shall be ConstructibleAsElement, as described in table [new table number] below. [*Note:* If the component is instantiated with a *scoped allocator* of type `A` (i.e., an allocator for which `is_scoped_allocator<A>` is true), then the `construct_element` function may pass an inner allocator argument to `T`'s constructor. – *end note*]

Rather than generalize all of the requirements on T for the entire container, 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 `is_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 the item type use an allocator (because allocator-specific behavior depends on the `uses_allocator` trait, which applies only to classes that use allocators).

In section [container.requirements] (23.1), add a new table before Table 87:

> In table *n*, A denotes an allocator, I denotes an allocator of type A::inner_allocator_type (if any), X denotes a container class containing objects of type T and instantiated with allocator type A, a and b denote values of type X, 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. Args denotes a template parameter pack; args denotes a function parameter pack with the pattern Args&&.

Table *n*: `ConstructibleAsElement<A,T,Args>` requirements

expression	post-condition
! (is_scoped_allocator<A>::value && uses_allocator<T,I>::value) || constructible_with_allocator_prefix<T, I, Args…>::value || constructible_with_allocator_suffix<T, I, Args…>::value	`true`

```
concept ConstructibleAsElement<class Alloc, class T, class... Args>
{
    requires Allocator<Alloc>;
    void construct_element(Alloc&, T&, Args...);
}

template <class Alloc, class T, class... Args>
  requires ScopedAllocator<Alloc> &&
    UsesAllocator<T, Alloc::inner_allocator_type> &&
    ConstructibleWithAllocatorPrefix<T, Alloc::inner_allocator_type,
                                     Args...>
concept_map ConstructibleAsElement<Alloc, T, Args...> {
    void construct_element(Alloc&, T&, Args...);
}

template <class Alloc, class T, class... Args>
  requires ScopedAllocator<Alloc> &&
    UsesAllocator<T, Alloc::inner_allocator_type> &&
    ConstructibleWithAllocatorSuffix<T, Alloc::inner_allocator_type,
                                     Args...>
concept_map ConstructibleAsElement<Alloc, T, Args...> {
    void construct_element(Alloc&, T&, Args...);
}

template <class Alloc, class T, class... Args>
  requires ! ScopedAllocator<Alloc> ||
    ! UsesAllocator<T, Alloc::inner_allocator_type>
concept_map ConstructibleAsElement<Alloc, T, Args...> {
    void construct_element(Alloc&, T&, Args...);
}
```

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

expression	return type	operational semantics	assertion/note pre/post-condition	complexity
`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

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.

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 with a `value_type` of T using allocator of type A, 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 A, Q is an allocator type.

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

expression	**return type**	**assertion/note pre/post-condition**	**complexity**
`allocator_type`	`A`	*requires:* `allocator_type::value_type` is the same as `X::value_type`.	compile time
`uses_allocator<X,Q>`	derived from `true_type` or `false_type`	`true_type` if `Q` is convertible to `A`	compile time
`Constructible_with_allocator_suffix<X>`	Derived from `true_type`		compile time
`get_allocator()`	`A`		constant
`X()` `X u;`		*requires:* DefaultConstructible<A>. *post:* `u.size() == 0`, `get_allocator()== A()`	constant
`X(m)` `X u(m);`		`post: u.size() == 0, get_allocator() == m`	constant
`X(t,m)` `X u(t,m);`		*requires:* ConstructibleAsElement<A,T,T> *post:* `u == t`, `get_allocator() == m`	linear

`X(rv,m)`
`X u(rv,m);`
	requires: ConstructibleAsElement<A,T,T&&>
	post: `u` shall be equal to the value that `rv` had before this construction, `get_allocator() == m`
		constant if m == rv.get_allocator(), else linear

Adds the allocator requirements. The `uses_allocator` trait is computed automatically. We specify the allocator-extended default, move, and copy constructors as allocator-suffix-style constructors and define the appropriate trait. Note that *all* containers will now have a constructor that takes a single allocator argument. The absence of such a constructor (e.g., for `map`) 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, A denotes X::allocator_type if it exists and std::allocator<T> if X does not have an allocator type, 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*: ConstructibleAsElement<A,T,Args>. Inserts an object of type T constructed with ~~T(~~std::forward<Args>(args)...~~)~~.~~;~~
`a.insert(p,t)`	`iterator`	*requires*: ~~T shall be CopyConstructible~~ ConstructibleAsElement<A,T,T> and CopyAssignable<T>. inserts a copy of t before p.
`a.insert(p,rv)`	`iterator`	*requires*: ConstructibleAsElement<A,T,T&&> and MoveAssignable<T>. inserts a copy of rv before p.
`a.erase(q)`	`iterator`	*requires:* MoveAssignable<T>. Erases the element pointed to by q
`a.erase(q1,q2)`	`iterator`	*requires:* MoveAssignable<T>. 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:* ConstructibleAsElement<A,T,Args>	`list, deque`

`a.push_-back(args)`	`void`	`a.emplace(a.end(), std::forward<Args>(args)…)` *requires:* ConstructibleAsElement<A,T,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` for some container types.

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-related traits of the Compare type and whether `is_scoped_allocator` is true for the container's allocator 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. A denotes the storage allocator used by X, if any, or std::allocator<X::value_type> otherwise, and m denotes an allocator of type convertible to A.

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

`X(c)`
`X a(c)`
		requires: ConstructibleAsElement<A,`key_compare`, `key_compare`>.
		constructs an empty container.
		uses a copy of c as a comparison object
			constant

`X()`
`X a;`
		requires: ConstructibleAsElement<A, `key_compare`>
		constructs an empty container
		uses Compare() as a comparison object
			constant

`X(i,j,c)`
`X a(i,j,c);`
	requires: ConstructibleAsElement<A,`key_compare`, `key_compare`>.
	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: ConstructibleAsElement<A, `key_compare`>
	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 the allocator-related traits of the Hash and Pred types and whether `is_scoped_allocator` is true for the container's allocator type.

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&);
```

And the following trait specialization:

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

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()` unless the copy constructor for `Allocator` throws.

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()`
`get_allocator()`	`alloc`

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 Alloc>
  struct constructible_with_allocator_suffix<deque<T, Alloc> >
    : true_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 Alloc>
  struct constructible_with_allocator_suffix<list<T, Alloc> >
    : true_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 Alloc>
  struct constructible_with_allocator_suffix<vector<T, Alloc> >
    : true_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 `constructible_with_allocator_suffix` 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*]

The ability to pass an allocator to an adaptor is important for allowing containers of adaptors.

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, class Alloc>
  struct uses_allocator<queue<T, Container>, Alloc>
    : uses_allocator<Container, Alloc>::type { };

template <class T, class Container>
  struct constructible_with_allocator_suffix<queue<T, Container> >
    : true_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 Compare&,
                                      const Alloc&);
template <class Alloc> priority_queue(const Compare&,
                                      const Container&,
                                      const Alloc&);
template <class Alloc> priority_queue(const Compare&,
                                      Container&&,
                                      const Alloc&);
template <class Alloc> priority_queue(priority_queue&&,
                                      const Alloc&);
```

And add the following trait specializations:

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

template <class T, class Container, class Compare >
  struct constructible_with_allocator_suffix<
      priority_queue<T, Container, Compare> >
    : true_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, class Alloc>
  struct uses_allocator<stack<T, Container>, Alloc>
    : uses_allocator<Container, Alloc>::type { };

template <class T, class Container>
  struct constructible_with_allocator_suffix<stack<T, Container> >
    : true_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 Alloc>
  struct constructible_with_allocator_suffix<
      map<Key,T,Compare,Alloc> >
    : true_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.

Add the following constructors:

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

And add the following trait specialization:

```
template <class Key, class T, class Compare, class Alloc>
  struct constructible_with_allocator_suffix<
      multimap<Key,T,Compare,Alloc> >
    : true_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.

Add the following constructors:

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

And add the following trait specialization:

```
template <class Key, class Compare, class Alloc>
  struct constructible_with_allocator_suffix<
      set<Key,Compare,Alloc> >
    : true_type { };
```

multiset 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.

Add the following constructors:

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

And add the following trait specialization:

```
template <class Key, class Compare, class Alloc>
  struct constructible_with_allocator_suffix<
      multiset<Key,Compare,Alloc> >
    : true_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>.

Add the following constructors:

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

And add the following trait specialization:

```
template <class Key,class T,class Hash,class Pred,class Alloc>
  struct constructible_with_allocator_suffix<
      unordered_map<Key,T,Hash,Pred,Alloc> >
    : true_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>.

Add the following constructors:

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

And add the following trait specialization:

```
template <class Key,class T,class Hash,class Pred,class Alloc>
  struct constructible_with_allocator_suffix<
      unordered_multimap<Key,T,Hash,Pred,Alloc> >
    : true_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.

Add the following constructors:

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

And add the following trait specialization:

```
template <class Value,class Hash,class Pred,class Alloc>
  struct constructible_with_allocator_suffix<
      unordered_set<Value,Hash,Pred,Alloc> >
    : true_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.

Add the following constructors:

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

And add the following trait specialization:

```
template <class Value,class Hash,class Pred,class Alloc>
  struct constructible_with_allocator_suffix<
      unordered_multiset<Value,Hash,Pred,Alloc> >
    : true_type { };
```

Function Changes

In section 20.5.15.2 [func.wrap.func], change the list of constructors as follows:

```
explicit function();
function(unspecified-null-pointer-type );
function(const function&);
function(function&&);
template<class F> function(F);
~~template<class F, class A> function(F, const A&);~~
template<class F> function(F&&);

template<class A> function(allocator_arg_t, const A&);
template<class A> function(allocator_arg_t, const A&,
                           unspecified-null-pointer-type );
template<class A> function(allocator_arg_t, const A&,
                           const function&);
template<class A> function(allocator_arg_t, const A&, function&&);
template<class F, class A> function(allocator_arg_t, const A&, F);
template<class F, class A> function(allocator_arg_t, const A&, F&&);
```

And add the following trait specializations:

```
template<class R, class... ArgTypes >
  struct constructible_with_allocator_prefix<
      function<R(ArgTypes...)> >
    : true_type { };

template<class R, class... ArgTypes, class Alloc>
  struct uses_allocator<function<R(ArgTypes...)>, Alloc >
    : true_type { };
```

```
template<class R, class... ArgTypes, class Alloc>
  concept_map UsesAllocator<function<R(ArgTypes...)>, Alloc > {
    typedef Alloc allocator_type;
}
```

A function object accepts an allocator using type erasure and thus has no `allocator_type` nested type. We adjust for that by explicitly specializing uses_allocator trait or mapping the UsesAllocator concept. The prefix-style allocator argument is chosen to avoid overload ambiguities between the A and F parameters.

Implementation Experience

All of the elements in this proposal have been implemented and most have been used extensively at Bloomberg LP and at other companies (e.g. Bear Stearns) for several years. We make frequent use of short-lived arena allocators and allocators that use special memory regions, and the scoped allocator semantics have provided a powerful way to manage memory. The Bloomberg interface is quite close to the interface presented in this proposal.

An open-source implementation that implements the exact interface from this proposal is under development. The non-concept-based interface can be implemented almost entirely using a C++03 compiler (though variadic templates must be simulated with fixed-length parameter lists).

There is also an implementation of an earlier version of this proposal by a commercial library vendor (Dinkumware).