Doc No: N2946=09-0136
Date: 2009-09-27
Author: Pablo Halpern
Intel Corp.
phalpern@halpernwightsoftware.com

Allocators post Removal of C++ Concepts

Contents

Motivation and Background

The adoption of N2554 (The Scoped Allocator Model) and N2525 (Allocator-specific Swap and Move Behavior) in Bellevue (February/March 2008) made allocators much more useful and flexible than they were in 1998. It has been pointed out, however, that these improvements came at the cost of some interface complexity. Of particular concern (expressed strongly in US 65 and US 74.1) is the fact that the presence of scoped allocators requires the definition and testing of traits in numerous places in the standard library and that the `pair` class template was made too complex by the addition of allocator-related constructors.

A couple of concepts-related papers (N2768 and N2840) attempted to simplify the use of allocators by moving most scoped-allocator knowledge into the scoped-allocator adaptor classes, and most allocator-propagation machinery into the Allocator concept. In addition, N2908 was on the verge of removing allocator interfaces from pair. But then concepts were dropped from the core language in Frankfurt (July 2009), rendering these proposals moot.

This paper attempts to recapture the simplifications from N2768 but without the use of concepts and even goes a step or two further towards simplifying both the use of allocators (within containers) and the definition of allocators. Since the time N2554 and N2525 were accepted, we have benefited from concept-oriented thinking as well as additional experience with variadic templates. Significantly-improved compiler support for variadic templates and extended SFINAE using `decltype` has allowed everything in this paper to be fully implemented and shown not only to work, but to present a reasonable and clean interface for container and allocator authors.

Issues and National Body Comments Addressed in this Paper

If accepted into the WP, this proposal should resolve the following issues and national-body comments:

Issues: 431, 580, 635, 1075, 1166, 1172

National body comments: US 65 and US 74.1 (except that the issues with pair have been split off into a separate paper, N2945).

Document Conventions

Any reference to section names and numbers are relative to the **pre-concepts, August 2008 WP**, N2723 (pre-San Francisco).

> Existing and proposed working paper text is indented and shown in dark blue. Small edits to the working paper are shown with ~~red strikeouts for deleted text~~ and green underlining for inserted text within the indented blue original text. Large proposed insertions into the working paper are shown in the same dark blue indented format (no green underline).

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

Requests for LWG opinions and guidance appear with light (yellow) shading. It is expected that any changes resulting from such guidance would be minor and would not impede acceptance of this paper in the same meeting.

Summary

The `allocator_traits` struct

The keystone of this proposal is the definition of an `allocator_traits` template containing types and static member functions for using allocators, effectively replacing the `Allocator` concept that was lost in Frankfurt. A container, `C<T,Alloc>` accesses all allocator functionality through `allocator_traits<Alloc>` rather than through the allocator itself. For example, to allocate *n* objects, a container would call:

```
auto p = allocator_traits<Alloc>::allocate(myalloc, n);
```

instead of

```
auto p = myalloc.allocate(n);
```

Like `iterator_traits`, `allocator_traits` provides an adaptation point for allocators. Although C++0x allocators have a richer interface than C++98 allocators, forward compatibility is maintained because `allocator_traits` provides default implementations for the new features. In addition, `allocator_traits` provides default implementations even for features that were present in 1998. The new allocator requirements, therefore, are smaller than they were in 1998, thus making allocators easier to write. The following comprises a minimalist allocator interface that meets the proposed new requirements:

```
template <typename Tp>
class SimpleAllocator
{
  public:
    typedef Tp value_type;

    template <typename T>
      struct rebind { typedef SimpleAllocator<T> other; };

    SimpleAllocator(ctor args);

    template <typename T> SimpleAllocator(const SimpleAllocator<T>& other);

    Tp* allocate(std::size_t n);
    void deallocate(Tp* p, std::size_t n);
};
```

Note the absence of `pointer` and `reference` types and `construct`, `destroy`, and `max_size` methods, which are now optional because `allocator_traits` provides defaults for these members. In addition, the allocator propagation functions (`select_on_container_copy_construction`, `on_container_copy_assignment`, `on_container_move_assignment`, and `on_container_swap`) are given default implementations in `allocator_traits`, simplifying most allocators and providing forward-compatibility between the C++98 interface and the C++0x. If new features are added to allocators in the future, allocator_traits will provide a convenient adaptor interface for forward compatibility.

Non-raw pointer types

One of changes made in N2768 was the removal of the weasel words that allowed an implementation to assume that an allocator's pointer is the same as value_type*. The Allocator concept in N2768 provided constraints for pointer that lifted this restriction. Allowing for pointer types other than value_type* (a.k.a. "fancy" pointers) is important for the use of shared memory, relocatable memory, and other interesting applications.

In this proposal, we restore the ability to use fancy pointers by specifying a minimum set of requirements for the pointer type. We also introduce a new void_pointer type that allows the construction of recursive data structures (e.g., trees and lists) without creating cycles in the declaration of the allocator pointer type.

The key requirements for an allocator's pointer type are that it has pointer-like syntax (i.e., it can be dereferenced using operator*), that it is convertible to the corresponding void_pointer, and that there exists a function template, rebind_pointer<T> for converting a void_pointer back into a pointer. If an allocator does not define a pointer type, allocator_traits will provide default types for pointer, const_pointer, void_pointer, and const_void_pointer of value_type*, const value*, void*, and const void*, respectively. The above pointer requirements were carefully crafted to be harmonious with the intent of N2913 (SCARY Iterator Assignment and Initialization).

Simplified traits and segregation of scoped-allocator functionality

US 65 reads:

> Scoped allocators and allocator propagation traits add a small amount of utility at the cost of a great deal of machinery. The machinery is user visible, and it extends to library components that don't have any obvious connection to allocators, including basic concepts and simple components like pair and tuple.

The problem being described is that the traits that were added to support scoped allocators and allocator propagation are too visible and too intrusive. Ideally, only users who want scoped allocators or want to create an allocator with non-default propagation semantics would need to pay attention to this machinery, and even then the machinery should be as simple as possible.

In this proposal, we address this issue in two ways: 1) the machinery necessary to build and use a scoped allocator is moved into the scoped_allocator_adaptor template and is no longer mentioned in the general container section. 2) the functions used for allocator propagation are simplified and given default implementations in the `allocator_traits` template. Finally, [N2945] addresses the problem with the explosion of `pair` constructors – again moving the interface out of `pair` and into `scoped_allocator_adaptor`.

In total, the following allocator-related type traits and template function are removed:

```
is_scoped_allocator,
constructible_with_allocator_prefix,
constructible_with_allocator_suffix,
allocator_propagate_never,
allocator_propagate_on_copy_construction,
allocator_propagate_on_move_assignment,
allocator_propagate_on_copy_assignment,
allocator_propagation_map

construct_element
```

Implementation experience

Everything in this proposal has been implemented with an eye towards making allocators as easy to use as possible. The main clients for the allocator interface are the container templates, hence it was necessary to implement at least one container in order to test the usability and implementability of the allocator interface. We chose to implement the `std::list` template because, being a node-based container, `list` best exercises the part of the interface that deals with fancy pointer types and rebound allocators. In the process, we discovered which interfaces were easy to use and which interfaces got in the way, and made adjustments. This proposal has thus been refined to reflect the most workable interface to date.

Our experience implementing the `list` template is that the `allocator_traits` interface is quite straight-forward to use. Using a few `typedef`s, the extra layer on top of the allocator is not at all cumbersome. Although there was some complexity in the implementation of `scoped_allocator_adaptor`, none of that complexity leaked into `list`. With this experience, we are confident that the ideas in this proposal represents a significant improvement over both C++98 allocators and the current working draft.

A complete implementation of `allocator_traits` and `scoped_allocator_adaptor`, as well as an implementation of `list` using `allocator_traits` is available at http://www.halpernwightsoftware.com/WG21/allocator_traits.tgz. (The implementation is tuned to the capabilities and limitations of gcc 4.4.1.)

Formal Wording

This wording is far from complete. I have included only the highlights, so far.

Header `<memory>` changes

Modify the top of section 20.7, header <memory> synopsis, as shown:

```
// 20.7.1, allocator argument tag
struct allocator_arg_t { };
const allocator_arg_t allocator_arg = allocator_arg_t();

// 20.7.2, uses_allocator
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;

// 20.7.3, allocation propagation traits
template <class Alloc> struct allocator_propagate_never;
template <class Alloc> struct allocator_propagate_on_copy_construction;
template <class Alloc> struct allocator_propagate_on_move_assignment;
template <class Alloc> struct allocator_propagate_on_copy_assignment;
template <class Alloc> struct allocator_propagation_map;

// 20.7.3 allocator traits
template <class Alloc> struct allocator_traits;

// 20.7.5, the default allocator:
template <class T> class allocator;
template <> class allocator<void>;
template <class T, class U>
  bool operator==(const allocator<T>&, const allocator<U>&) throw();
template <class T, class U>
  bool operator!=(const allocator<T>&, const allocator<U>&) throw();

// 20.7.6, scoped allocator adaptor
template <class OuterAlloc, class... InnerAllocs> = 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 OuterA1, class OuterA2, class... InnerAllocs>
  bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...1>& a,)
                  const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b);
template <class OuterA1, class OuterA2, class... InnerAllocs>
  bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerAllocs...1>& a,)
                  const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b);
```

```
// 20.7.7, raw storage iterator:
template <class OutputIterator, class T> class raw_storage_iterator;

// 20.7.8, temporary buffers:
template <class T>
  pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
template <class T>
  void return_temporary_buffer(T* p);

// 20.7.9, construct element
template <class Alloc, class T, class... Args>
  void construct_element(Alloc& alloc, T& r, Args&&... args);

// 20.7.10, specialized algorithms:
template <class T> T* addressof(T& r);
template <class T> T* addressof(T&& r);
template <class T> T       * pointer_rebind(void *p);
template <class T> T const* pointer_rebind(void const *p);
template <class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                     ForwardIterator result);
template <class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
                                       ForwardIterator result);
template <class ForwardIterator, class T>
  void uninitialized_fill(ForwardIterator first, ForwardIterator last,
                          const T& x);
template <class ForwardIterator, class Size, class T>
  void uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
```

The `addressof` and `pointer_rebind` function templates

In section 20.7.10 [specialized.algorithms], insert the following:

```
template <class T> T* addressof(T& r);
template <class T> T* addressof(T&& r);
```

Returns: The actual address of the object referenced by `r`, even in the presence of an overloaded `operator&`.

Throws: nothing.

This function is useful in its own right but is required for describing and implementing a number of allocator features. An implementation can be found in the boost library and in the sample implementation described in the introduction.

Note to the editor: This function was originally added in San Francisco, but was part of a concepts paper and was most likely removed when concepts were removed. This non-concept version removes the second overload, as per the resolution of issue 970.

```
template <class T> T*       pointer_rebind(void *p);
template <class T> T const* pointer_rebind(void const *p);
```

Precondition: `T` is an (optionally cv-qualified) object type.

Returns: `static_cast<T*>(p).`

Throws: nothing.

Remarks: A program may overload `pointer_rebind` for a user-defined pointer-like type or template, in the namespace of that type or template, for the purpose of converting from a generic pointer to a pointer specific to `T`, as required by the allocator requirements for `pointer`, `const_pointer`, `void_pointer`, and `const_void_pointer` (see allocator requirements, 20.1.2).

[*Example:*

```
namespace mine {
  template <class T> MyPointer { ... };
  typedef MyPointer<void> MyVoidPointer;
  typedef MyPointer<const void> MyConstVoidPointer;

  template <class T>
    MyPointer<T> pointer_rebind(MyVoidPointer p);
  template <class T>
    MyPointer<const T> pointer_rebind(MyConstVoidPointer p);
}
```

– end example]

Allocator Requirements

Modify section 20.1.2 [allocator.requirements], as follows:

The library describes a standard set of requirements for allocators, which are class-type objects that encapsulate the information about an allocation model. This information includes the knowledge of pointer types, the type of their difference, the type of the size of objects in this allocation model, as well as the memory allocation and deallocation primitives for it. All of the string types (Clause 21) and containers (Clause 23) except array are parameterized in terms of allocators.

Table 39 describes the requirements on types manipulated through allocators. ~~All the operations on the allocators are expected to be amortized constant time.~~ Table 40 describes the requirements on allocator types. The template class `allocator_traits` ([allocator.traits]) supplies a uniform interface to all allocator types. Those expressions that have a default value in table 40 may be omitted from an allocator class and will be supplied by the `allocator_traits` instantiation for that class.

Table 39 – Descriptive variable definitions

Variable	Definition
`T, U, C`	any non-const, non-reference object type
`V`	a type convertible to `T`
`X`	an Allocator class for type `T`
`Y`	the corresponding Allocator class for type `U`
`XX`	The type `allocator_traits<X>`
`YY`	The type `allocator_traits<Y>`
`t`	a value of type `const T&`

a, a1, a2	values of type X&
a3	rvalue of type X
b	a value of type Y
c	a dereferenceable pointer of type C*
p	a value of type ~~X~~XX::pointer_type, obtained by calling a1.allocate, where a1 == a
q	a value of type ~~X~~XX::const_pointer_type obtained by conversion from a value p
w	a value of type XX::void_pointer obtained by conversion from a value p
z	a value of type XX::const_void_pointer obtained by conversion from a value q or a value w
r	a value of type ~~X::reference~~T& obtained by the expression *p.
s	a value of type ~~X::const_reference~~const T&~~-~~obtained by the expression *q or by conversion from a value r.
u	a value of type ~~Y~~YY::const_pointer obtained by calling ~~Y~~YY::allocate, or else ~~0~~nullptr.
v	a value of type V
n	a value of type ~~X~~XX::size_type
Args	a template parameter pack
args	a function parameter pack with the pattern Args&&

Table 40 – Allocator requirements

Expression	Return type	Assertion/note pre-/post-condition	Default
X::pointer	Pointer to T		T*
X::const_pointer	Pointer to const T	X::pointer is convertible to X::const_pointer	T const*
X::void_pointer Y::void_pointer	generic pointer type	X::pointer is convertible to X::void_pointer. X::void_pointer and Y::void_pointer are the same type.	void*
X::const_void_pointer Y::const_void_pointer	generic const pointer type	X::pointer and X::const_pointer are convertible to X::const_void_pointer. X::const_void_pointer and Y::const_void_pointer are the same type.	void const*
~~X::reference~~	~~T&~~		
~~X::const_reference~~	~~T const&~~		
X::value_type	Identical to T		
X::size_type	unsigned integral type	a type that can represent the size of the largest object in the allocation model.	size_t
X::difference_type	signed integral type	a type that can represent the	ptrdiff_t

		difference between any two pointers in the allocation model.	
`typename X::template rebind<U>::other`	`Y`	For all `U` (including `T`), `Y::template rebind<T>::other` is `X`.	
`*p`	`T&`		
`*q`	`T const&`	`*q` refers to the same object as `*p`	
`p.operator->()`	`T*`	equivalent to `addressof(*p)`	
`q.operator->()`	`T const*`	equivalent to `addressof(*q)`	
`pointer_rebind<T>(w)`	`X::pointer`	`pointer_rebind<T>(w) == p`	
`pointer_rebind<const T>(z)`	`X::const_pointer`	`pointer_rebind<const T>(z) == q`	
`pointer d(nullptr); const_pointer e(nullptr);`		`d` and `e` are null pointers and need not be dereferenceable, `!d != false`, `!e != false`	
`void_pointer d(nullptr); const_void_pointer e(nullptr);`		`d` and `e` are null pointers need not be dereferenceable `d` and `e` are null pointers and need not be dereferenceable, `!d != false`, `!e != false`	
`!p`	convertible to `bool`	true if `p` is a null pointer, else false	
`!q`	convertible to `bool`	true if `q` is a null pointer, else false	
`!w`	convertible to `bool`	true if `w` is a null pointer, else false	
`!z`	convertible to `bool`	true if `z` is a null pointer, else false	
`a.address(r)`	`X::pointer`		`addressof(r)`
`a.address(s)`	`X::const_pointer`		`addressof(s)`
`a.allocate(n)` ~~`a.allocate(n,u)`~~	`X::pointer`	X::pointer Memory is allocated for n objects of type T but objects are not constructed. allocate may raise an appropriate exception. The result is a random access iterator.[227] [*Note*: If n == 0, the return value is unspecified. — *end note*]	
`a.allocate(n,u)`	`X::pointer`	Same as `a.allocate(n)`. The use of `u` is unspecified, but intended as an aid to locality if an implementation so desires.	`a.allocate(n)`
`a.deallocate(p,n)`	(not used)	All n T objects in the area pointed to by p shall be destroyed prior to this call. n shall match the value passed to allocate to obtain this memory. Does not throw exceptions. [*Note*: p shall not be ~~null~~singular .— *end note*]	
`a.max_size()`	`X::size_type`	the largest value that can meaningfully be passed to `X::allocate()`	`numeric_limits<size_type>::max()`

a1 == a2	bool	returns true iff storage allocated from each can be deallocated via the other. operator== shall be reflexive, symmetric, and transitive.	
a1 != a2	bool	same as !(a1 == a2)	
a1 == b	bool	same as a == Y::rebind<T>::other(b)	
a1 != b	bool	same as !(a1 == b)	
~~X()~~		~~creates a default instance. [*Note*: a destructor is assumed. — *end note*]~~	
X a1(a);		post: a1 == a	
X a(b);		post: Y(a) == b, a == X(b)	
a.construct(~~p~~c,args)	(not used)	Effect: Constructs an object of type ~~T~~C at ~~p~~c ~~by invoking T(forward<Args>(args)...)~~	new ((void*)c) C(forward<Args>(args)...)
a.destroy(~~p~~c)	(not used)	Effect: Destroys the object at ~~p~~c	c->~T()
a.select_on_container_copy_construction()	X	Typically returns either a or X()	return a;
a.on_container_copy_assignment(a1)	convertible to bool	Effect: typically either does nothing or assigns a = a1. Returns: true if a was modified, else false.	return false;
a.on_container_move_assignment(a3)	convertible to bool	Effect: typically either does nothing or assigns a = a3. Returns: true if a was modified, else false. Must not throw.	return false;
a.on_container_swap(a1)	convertible to bool	Effect: typically either does nothing or swaps a with a1. Returns: true if a or a1 was modified, else false. Must not throw.	return false;

The X::pointer, X::const_pointer, X::void_pointer, and X::const_void_pointer types shall satisfy the requirements of EqualityComparable, DefaultConstructible, CopyConstructible, CopyAssignable, Swappable, and Destructible (20.1.1 [utility.arg.requirements]). No constructor, comparison operator, copy operation, or swap operation on these types shall throw an exception. A default initialized object may have a singular value. X::pointer and X::const_pointer shall also satisfy the requirements for a random-access iterator (24.1 [iterator.requirements]).

The key changes from the WP are:

1. The addition of the void_pointer and const_void_pointer types and the rules defining the minimal set of operations on pointer types.

2. The addition of default values, especially for the new features.

3. The first argument to construct and destroy is now a pointer to arbitrary type, rather than a pointer-to-T. This change facilitates constructing objects in node-based containers where the value_type is different from the node type.

4. The addition of the allocator propagation functions.

Note that there is no `select_on_container_move_construction()` function. After some consideration, we decided that a move construction operation for containers must be constant-time and not throw, as per issue 1166. However, we disagree with the proposed resolution of 1166 wrt move assignment. Having move assignment silently move the allocator breaks C++98 compatibility. The reason is that move assignment can be invoked with no code changes in code that formerly used copy-assignment. In C++98 there was an effective guarantee that the allocator for a container never changes over the lifetime of the object. Thus, not only must there be a choice to not propagate the allocator on move assignment, it must be the default. There is no loss of efficiency, however for the typical stateless allocator and authors of stateful allocators can choose to make their allocators move on move assignment.

The member class `template rebind` in the table above is effectively a typedef template: if the name Allocator is bound to `SomeAllocator<T>`, then `Allocator::rebind<U>::other` is the same type as `SomeAllocator<U>`.

An allocator may constrain the types on which it may be instantiated or on which its `construct` member may be called. If a type cannot be used with a particular allocator, the allocator or call to `construct` will fail to instantiate.

[*Example:* The following is an allocator class template supporting the minimal interface that satisfies the requirements in Table 40:

```
template <typename Tp>
class SimpleAllocator
{
  public:
    typedef Tp value_type;

    template <typename T>
      struct rebind { typedef SimpleAllocator<T> other; };

    SimpleAllocator(ctor args);

    template <typename T> SimpleAllocator(const SimpleAllocator<T>& other);

    Tp* allocate(std::size_t n);
    void deallocate(Tp* p, std::size_t n);
};
```

– end example]

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

~~— 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. In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in Table 40 are implementation-defined.~~

The weasel words are gone. Raise your glass and make a toast.

If the alignment associated with a specific over-aligned type is not supported by an allocator, instantiation of the allocator for that type may fail. The allocator also may silently ignore the requested alignment. [*Note*: additionally, the member function allocate for that type may fail by throwing an object of type `std::bad_alloc`.— *end note*]

Allocator-related traits

Completely replace section 20.7.2 [allocator.traits] with the following [uses.allocator] section:

20.7.2 uses_allocator [users.allocator] ~~Allocator-related traits [allocator.traits]~~

```
template <class T, class Alloc> struct uses_allocator;
```

Remark: Automatically detects whether `T` has a nested `allocator_type` that is convertible from `Alloc`. Meets the `BinaryTypeTrait` requirements (20.5.1). An instantiation will be derived from `true_type` if a type `T::allocator_type` exists and `is_convertible<Alloc, T::allocator_type>::value != false`. A program may specialize this ~~type~~`struct` to derive from `true_type` for a user-defined type `T` that does not have a nested `allocator_type` but is nonetheless constructible using the specified `Alloc`. Otherwise, this `struct` will be derived from `false_type`.

~~Remark: uses_allocator<T, Alloc> shall be derived from true_type if Convertible<Alloc, T::allocator_type>, otherwise derived from false_type.~~

~~The class templates `is_scoped_allocator`, `constructible_with_allocator_suffix`, and `constructible`, ...~~ [rest of section removed]

Completely delete section 20.7.3 [allocator.propagation]:

~~**20.7.3 Allocator propagation traits [allocator.propagation]**~~

~~Etc.~~

Insert a new allocator traits section:

20.7.3 Allocator traits [allocator.traits]

```
namespace std {
  template <typename Alloc> struct allocator_traits {

    typedef Alloc allocator_type;

    typedef typename Alloc::value_type value_type;

    typedef see below pointer;
    typedef see below const_pointer;
    typedef see below void_pointer;
    typedef see below const_void_pointer;

    typedef see below difference_type;
```

```
    typedef see below size_type;

    template <typename T> using rebind_alloc =
      typename Alloc::template rebind<T>::other;
    template <typename T> using rebind_traits =
      allocator_traits<rebind_alloc<T> >;

    static pointer allocate(Alloc& a, size_type n);
    static pointer allocate(Alloc& a, size_type n, const_void_pointer hint);

    static void deallocate(Alloc& a, pointer p, size_type n);

    template <typename T, typename... Args>
      static void construct(Alloc& a, T* p, Args&&... args);

    template <typename T>
      static void destroy(Alloc& a, T* p);

    static size_type max_size(const Alloc& a);

    static pointer       address(const Alloc& a, value_type& r);
    static const_pointer address(const Alloc& a, const value_type& r);

    static Alloc select_on_container_copy_construction(const Alloc& rhs);
    static bool on_container_copy_assignment(Alloc& lhs, const Alloc& rhs);
    static bool on_container_move_assignment(Alloc& lhs, Alloc&& rhs);
    static bool on_container_swap(Alloc& lhs, Alloc& rhs);
  };
}
```

20.7.3.1 Allocator traits type members

`typedef` *see below* `pointer;`

Type: `Alloc::pointer` if such a type exists, otherwise `value_type*`.

`typedef` *see below* `const_pointer;`

Type: `Alloc::const_pointer` if such a type exists, otherwise `const value_type*`.

`typedef` *see below* `void_pointer;`

Type: `Alloc::void_pointer` if such a type exists, otherwise `void*`.

`typedef` *see below* `const_void_pointer;`

Type: `Alloc::const_void_pointer` if such a type exists, otherwise `void*`.

`typedef` *see below* `difference_type;`

Type: `Alloc::difference_type` if such a type exists, otherwise `ptrdiff_t`.

`typedef` *see below* `size_type;`

Type: `Alloc::size_type` if such a type exists, otherwise `size_t`.

20.7.3.2 Allocator traits static member functions

`static pointer allocate(Alloc& a, size_type n);`

Returns: `a.allocate(n).`

```
static pointer allocate(Alloc& a, size_type n, const_void_pointer hint);
```

Returns: `a.allocate(n, hint)` if such an expression would be well formed, otherwise `a.allocate(n)`.

```
static void deallocate(Alloc& a, pointer p, size_type n);
```

Effects: calls `a.deallocate(p, n)`.

```
template <typename T, typename... Args>
  static void construct(Alloc& a, T* p, Args&&... args);
```

Effects: calls `a.construct(p, std::forward<Args>(args)... )` if such a call would be well formed, otherwise invokes `new (static_cast<void*>(p)) T(std::forward<Args>(args)...)`.

```
template <typename T>
  static void destroy(Alloc& a, T* p);
```

Effects: calls `a.destroy(p)` if such a call would be well formed, otherwise invokes `p->~T()`.

```
static size_type max_size(const Alloc& a);
```

Returns: `a.max_size()` if such a call would be well-formed, otherwise `numeric_limits<size_type>::max()`.

```
static pointer       address(const Alloc& a, value_type& r);
static const_pointer address(const Alloc& a, const value_type& r);
```

Returns: `a.address(r)` if such a call would be well formed, otherwise `std::addressof(r)`.

```
static Alloc select_on_container_copy_construction(const Alloc& rhs);
```

Returns: `rhs.select_on_container_copy_construction()` if such a call would be well formed, otherwise `rhs`.

```
static bool on_container_copy_assignment(Alloc& lhs, const Alloc& rhs);
```

Returns: `lhs.on_container_copy_assignment(rhs)` if such a call would be well formed, otherwise false.

```
static bool on_container_move_assignment(Alloc& lhs, Alloc&& rhs);
```

Returns: `lhs.on_container_move_assignment(std::move(rhs))` if such a call would be well formed, otherwise false.

Throws: nothing

Should the `rhs` argument be an rvalue-reference? On the one hand, this is a move operation. On the other hand, the `rhs` is a member of a larger object. In order to call this function with an rvalue reference, the caller would need to write `traits::on_container_move_assignment(this->alloc, move(other.alloc))`.

```
static bool on_container_swap(Alloc& lhs, Alloc& rhs);
```

Returns: `lhs.on_container_swap(rhs)` if such a call would be well formed, otherwise false.

Throws: nothing

Completely delete section 20.7.9 [construct.element]:

~~20.7.9 construct_element [construct.element]~~

~~Etc.~~

Scoped allocator adaptors

Completely replace section 20.7.6 [allocator.adaptor] with the following:

20.7.6 Scoped allocator adaptor [allocator.adaptor]

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 passed to the constructor of every element within the container. This adaptor is instantiated with one outer and zero or more inner allocator types. If instantiated with only one allocator type the inner allocator becomes the `scoped_allocator_adaptor` itself, thus using the same allocator resource for the container and every element in the container, and, if the elements are themselves containers, each of their elements recursively. If instantiated with more than one allocator, the first allocator is the outer allocator for use by the container, the second allocator is passed to the constructors of the container's elements, and, if the elements are themselves containers, the third allocator is passed to the elements' elements, etc.. If containers are nested to a depth greater than the number of allocators, then the last allocator is used repeatedly, as in the single-allocator case, for any remaining recursions. [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 OuterAlloc, typename... InnerAllocs>
  class scoped_allocator_adaptor : public OuterAlloc
  {
    typedef allocator_traits<OuterAlloc>     OuterTraits;  // exposition only
    scoped_allocator_adaptor<InnerAllocs...> inner;        // exposition only

  public:
    typedef OuterAlloc                                outer_allocator_type;
    typedef see below                                  inner_allocator_type;

    typedef typename OuterTraits::size_type          size_type;
    typedef typename OuterTraits::difference_type    difference_type;
    typedef typename OuterTraits::pointer            pointer;
    typedef typename OuterTraits::const_pointer      const_pointer;
    typedef typename OuterTraits::void_pointer       void_pointer;
    typedef typename OuterTraits::const_void_pointer const_void_pointer;
    typedef typename OuterTraits::value_type         value_type;

    template <typename Tp>
    struct rebind {
      typedef scoped_allocator_adaptor<
        OuterTraits::template rebind_alloc<Tp>, InnerAllocs...> other;
    };

    scoped_allocator_adaptor();
    template <typename OuterA2>
      scoped_allocator_adaptor(OuterA2&& outerAlloc,
                               const InnerAllocs&... innerAllocs);
```

```
  scoped_allocator_adaptor(const scoped_allocator_adaptor& other);

  template <typename OuterA2>
    scoped_allocator_adaptor(const scoped_allocator_adaptor<OuterA2,
                                                InnerAllocs...>& other);
  template <typename OuterA2>
    scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2,
                                                 InnerAllocs...>&& other);

  ~scoped_allocator_adaptor();

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

  pointer       address(value_type& x)       const;
  const_pointer address(const value_type& x) const;

  pointer allocate(size_type n);
  pointer allocate(size_type n, const_void_pointer hint);
  void deallocate(pointer p, size_type n);
  size_type max_size() const;

  template <typename T, typename... Args>
    void construct(T* p, Args&&... args);
```

If N2926 is accepted, we will add:

```
  // Specializations to pass inner_allocator to pair::first and pair::second
  template <class T1, class T2>
    void construct(std::pair<T1,T2>* p);
  template <class T1, class T2, class U, class V>
    void construct(std::pair<T1,T2>* p, U&& x, V&& y);
  template <class T1, class T2, class U, class V>
    void construct(std::pair<T1,T2>* p, const std::pair<U, V>& pr);
  template <class T1, class T2, class U, class V>
    void construct(std::pair<T1,T2>* p, std::pair<U, V>&& pr);
```

```
  template <typename T>
    void destroy(T* p);

  // Allocator propagation functions.
  static scoped_allocator_adaptor
   select_on_container_copy_construction(const scoped_allocator_adaptor& rhs);

  bool on_container_copy_assignment(const scoped_allocator_adaptor& rhs);
  bool on_container_move_assignment(scoped_allocator_adaptor& rhs);
  bool on_container_swap(scoped_allocator_adaptor& other);
};

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

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

20.7.6.1 scoped_allocator_adaptor inner allocator type [allocator.adaptor.inner]

`typedef` *see below* `inner_allocator_type;`

> *Type*: If `sizeof...InnerAllocs` is zero, `scoped_allocator_adaptor<OuterAlloc>`, otherwise `scoped_allocator_adaptor<InnerAllocs...>`

20.7.6.2 scoped_allocator_adaptor constructors [allocator.adaptor.cntr]

```
scoped_allocator_adaptor();
```

> *Effects:* default-initializes the `OuterAlloc` base class and the `inner` allocator object

```
template <typename OuterA2>
  scoped_allocator_adaptor(OuterA2&& outerAlloc,
                           const InnerAllocs&... innerAllocs);
```

> *Effects:* initializes the `OuterAlloc` base class with `std::forward<OuterA2>(outerAlloc)` and `inner` with `innerAllocs...` (hence recursively initializing each allocator within the adaptor with the corresponding allocator from the argument list).

Note that we cannot forward innerAllocs because it is not in a deduced context and cannot, therefore, use perfect forwarding.

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

> *Effects:* initializes each allocator within the adaptor with the corresponding allocator from `other`.

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

> *Effects:* initializes each allocator within the adaptor with the corresponding allocator from `other`.

```
template <typename OuterA2>
  scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2,
                                                    InnerAllocs…>&& other);
```

> *Effects:* initializes each allocator within the adaptor with the corresponding allocator rvalue from `other`.

20.7.6.3 scoped_allocator_adaptor members [allocator.adaptor.members]

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

> *Returns:* if `sizeof...InnerAllocs` is zero, `*this`, else `inner`

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

> *Returns:* `static_cast<Outer&>(*this)` or `static_cast<Outer const&>(*this)`, respectively.

```
pointer       address(value_type& x)       const;
```

```
const_pointer address(const value_type& x) const;
```

Returns: `allocator_traits<OuterAlloc>::address(outer_allocator(),x)`

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

Returns: `allocator_traits<OuterAlloc>::allocate(outer_allocator(),n)`

```
pointer allocate(size_type n, const_void_pointer hint);
```

Returns: `allocator_traits<OuterAlloc>::allocate(outer_allocator(),n,hint)`

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

Effects: `allocator_traits<OuterAlloc>::deallocate(outer_allocator(),p,n)`

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

Returns: `allocator_traits<OuterAlloc>::max_size(outer_allocator())`

```
template <typename T, typename... Args>
  void construct(T* p, Args&&... args);
```

Effects: let *`OUTERMOST(x)`* be `x` if `x` does not have an `outer_allocator()` method, and *`OUTERMOST(x.outer_allocator())`* otherwise. If `uses_allocator<T,inner_allocator_type>::value` is not false and the expression `T(allocator_arg, inner_allocator(), std::forward<Args>(args)...)` is well formed, then calls *OUTERMOST(*`*this`*)*`.construct(p, allocator_arg, inner_allocator, std::forward<Args>(args)...)`. Otherwise, if `uses_allocator<T,inner_allocator_type>::value` is not false and the expression `T(std::forward<Args>(args)..., inner_allocator())` is well formed, then calls *OUTERMOST(*`*this`*)*`.construct(p, std::forward<Args>(args)..., inner_allocator())`. Otherwise, if `uses_allocator<T,inner_allocator_type>::value` is false, call *OUTERMOST(*`*this`*)*`.construct(p, std::forward<Args>(args)...)`. Otherwise the instantiation is ill formed. [*Note:* an error will result if `uses_allocator` evaluates true but the specific constructor does not take an allocator. This definition prevents a silent failure to pass an inner allocator to a contained element. – *end note*]

```
template <typename T>
  void destroy(T* p);
```

Effects: calls `outer_allocator().destroy(p)`

```
static scoped_allocator_adaptor
  select_on_container_copy_construction(const scoped_allocator_adaptor& rhs);
```

Returns: a new `scoped_allocator_adaptor` where each allocator in the adaptor is initialized from the result of calling `select_on_container_copy_construction` on the corresponding allocator in `rhs`.

```
bool on_container_copy_assignment(const scoped_allocator_adaptor& rhs);
```

Effects: For each allocator in the adaptor, calls `on_container_copy_assignment`, passing it the corresponding allocator in `rhs`.

Returns: true if any of the calls to `on_container_copy_assignment` returned true

```
bool on_container_move_assignment(scoped_allocator_adaptor& rhs);
```

Effects: For each allocator in the adaptor, calls `on_container_move_assignment`, passing it the corresponding allocator in `rhs`.

Returns: true if any of the calls to `on_container_move_assignment` returned true

```
bool on_container_swap(scoped_allocator_adaptor& other);
```

Effects: For each allocator in the adaptor, calls `on_container_swap`, passing it the corresponding allocator in `rhs`.

Returns: true if any of the calls to `on_container_swap` returned true

Changes to container and string wording

Change section 23.1.1 [container.requirements.general], paragraphs 3 and 4 as follows:

3 For the components defined in this clause that declare an `allocator_type`, oObjects stored in these components shall be constructed using ~~construct_element (20.7.9)~~the `allocator_traits<allocator_type>::construct` function and destroyed using the `allocator_traits<allocator_type>::destroy` function (20.7.3.2 [allocator.traits.funcs]). These `construct` and `destroy` functions are called only for the container's element type, not for internal types used by the container. [*Note:* This means, for example, that a node-based container might need to construct nodes containing aligned buffers, the call `construct` to place the element into the buffer. – *end note*] ~~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 89. [Note: If the component is instantiated with a scoped allocator of type A (i.e., an allocator for which is_scoped_allocator<A>::value is true), then construct_element may pass an inner allocator argument to T's constructor. —end note]~~

~~4 In table 89, T denotes an object type, A denotes an allocator, I denotes an allocator of type A::inner_allocator_type (if any),and Args denotes a template parameter pack~~

Delete table 89:

~~Table 89 — ConstructibleAsElement<A, T, Args> requirements [constructibleaselement]~~

~~Etc.~~

Modify the notes after Table 90 as follows:

Notes: the algorithms `swap()`, `equal()` and `lexicographical_compare()` are defined in Clause 25. Those entries marked "(Note A)" or "(Note B)" should have constant complexity. ~~Those entries marked "(Note B)" have constant complexity unless allocator_propagate_never<X::allocator_type>::value is true, in which case they have linear complexity.~~ Those entries marked "(Note C)" have constant complexity if `allocator_traits<allocator_type>::on_container_move_assignment` returns true or if `a.get_allocator() == rv.get_allocator()` ~~or if either allocator_propagate_on_move_assignment<X::allocator_type>::value is true or allocator_propagate_on_copy_assignment<X::allocator_type>::value is true~~ and linear complexity otherwise.

Modify Section 23.1.1 [container.requirements.general], paragraph 9:

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

In table 92 (Allocator-aware container requirements), modify selected rows as shown:

~~Constructible_with_allocator_suffix<X>~~	~~derived from true_type~~		~~compile time~~
…			
`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()`, otherwise linear

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

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

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

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

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

Unless described below, all operations have the same requirements and semantics as the primary vector template, except that operations dealing with the bool value type map to bit values in the container storage and `allocator_traits::construct` (20.7.3.2) is not used to construct these values.

Interaction with N2913

Care was taken in this proposal to be compatible with N2913 (SCARY Iterator Assignment and Initialization). If N2913 is accepted, the following minor changes would be needed:

1. Add `void_pointer` to the list of types on which an iterator may depend.

2. Add `void_pointer` and `const_void_pointer` to the list of types on which a const_iterator may depend.

Acknowledgements

Thanks to John Lakos, Howard Hinnant, Alisdair Merideth, Daniel Krügler, Steve Breitstein, and Mike Giroux for their help in crafting and reviewing this paper.

References

Documents referenced below can be found at http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2008/.

N2768: Allocator Concepts, part 1 (revision 2)
N2554: The scoped allocator model (Rev 2)
N2525: Allocator-specific move and swap

Documents referenced below can be found at http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2009/.

N2840: Defects and Proposed Resolutions for Allocator Concepts (Rev 2)
N2913: SCARY Iterator Assignment and Initialization
N2945: Proposal to Simplify pair (Rev 2)