Doc No: N2982=09-0172
Date: 2009-10-22
Author: Pablo Halpern
Intel Corp.
phalpern@halpernwightsoftware.com

Allocators post Removal of C++ Concepts (Rev 1)

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.

Changes from N2946

- Changed base document to most recent WP (N2960). Performed a sweep of the WP to find incorrect use of allocators or allocator concepts.
- Replaced the `pointer_rebind` mechanism with Howard Hinnant's simpler and more general `pointer_traits` mechanism.
- Added a default implementation for allocator_traits::rebind.
- Changed most allocator propagation traits to simple true_type/false_type typedefs.
- Nothrow requirement added to allocator copying and equality comparison.
- Added rigor to the requirements for `Allocator::pointer` types, `Allocator::rebind`, and `uses_allocator`.
- Numerous bug fixes, clarifications, rewordings, etc., thanks to my reviewers.

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: UK 241, US 65, US 77 and US 74.1 (except that the issues with pair have been split off into a separate paper, N2981, a minor revision of N2945).

Document Conventions

Any reference to section names and numbers are relative to the September 2009 WP, **N2960**.

> 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);
```

In the same way that `iterator_traits` provides an adaptation point for iterators, `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 list of non-optional 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 <class Tp>
class SimpleAllocator
{
  public:
    typedef Tp value_type;

    SimpleAllocator(ctor args);

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

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

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

Because certain members, like `construct` and `destroy`, are now optional, an implementation (with care) can detect their absence (and the absence of a specialization of `allocator_traits`) and perform certain optimizations. For example, copy-constructing the elements of a `vector<PodType, SimpleAllocator<PodType>>` could be done using `memcpy` because `SimpleAllocator` does not have a customized `construct` function.

Generalized 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 generalized 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 generalized 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 implicitly convertible to the corresponding `void_pointer` and explicitly convertible from the corresponding `void_pointer`, and that there exists a specialization of the `pointer_traits` class template, which describes a number of key attributes of the pointer type. 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. In addition, N2945 (revised by D2981) 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 generalized 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`, `pointer_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_rev1.tgz. (The implementation is tuned to the capabilities and limitations of gcc 4.4.1.)

Formal Wording

Header <`memory`> changes

Modify the top of section 20.8, header <memory> synopsis, as shown:

```
// 20.8.1, allocator argument tag
struct allocator_arg_t { };
constexpr allocator_arg_t allocator_arg = allocator_arg_t();

// 20.8.2, 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;~~

~~// 20.8.3, 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;~~

// 20.8.3 pointer traits
template <class Ptr> struct pointer_traits;
template <class T> struct pointer_traits<T*>;

// 20.8.4 allocator traits
template <class Alloc> struct allocator_traits;

// 20.8.5, 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();

// 20.8.6, 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, InnerA~~1~~llocs...~~1~~>& a,~~)~~
                  const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b);
template <class OuterA1, class OuterA2, class... InnerAllocs>
  bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerA~~1~~llocs...~~1~~>& a,~~)~~
                  const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b);

// 20.8.7, raw storage iterator:
template <class OutputIterator, class T> class raw_storage_iterator;

// 20.8.8, temporary buffers:
template <class T>
  pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
template <class T>
  void return_temporary_buffer(T* p);

~~// 20.8.9, construct element~~
~~template <class Alloc, class T, class... Args>~~
~~  void construct_element(Alloc& alloc, T& r, Args&&... args);~~
```

```
// 20.8.10, specialized algorithms:
template <class T> T* addressof(T& r);
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` function template

In section 20.8.10 [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 removed when concepts were removed. This non-concept version removes the second overload, as per the resolution of issue 970.

Allocator Requirements

Modify section 20.2.2 [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), containers (except array (Clause 23)), string buffers and string streams (Clause 27), and match_results (Clause 28) are parameterized in terms of allocators.

The template struct `allocator_traits` ([allocator.traits]) supplies a uniform interface to all allocator types. 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 and thus on types used to instantiate `allocator_traits`. A requirement is *optional* if the last column of Table 40 specifies a default for a given expression. Within the standard library `allocator_traits` template, an optional requirement that is not supplied by an allocator is replaced by the specified default expression. A user specialization of `allocator_traits` may provide different defaults, and may provide defaults for different requirements, than the primary template. Within Tables 39 and 40, the use of `move` and `forward` always refer to `std::move` and `std::forward`, respectively.

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`, obtained by calling `a1.allocate`, where `a1 == a`
`q`	a value of type ~~X~~XX`::const_pointer` 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`	`pointer_traits<X::pointer>::rebind<const T>`
`X::void_pointer` `Y::void_pointer`		`X::pointer` is convertible to `X::void_pointer`. `X::void_pointer` and	`pointer_traits<X::pointer>::rebind<void>`

		`Y::void_pointer` are the same type.	
`X::const_void_pointer` `Y::const_void_pointer`		`X::pointer`, `X::const_pointer`, and `X::void_pointer` are all convertible to `X::const_void_pointer`. `X::const_void_pointer` and `Y::const_void_pointer` are the same type.	`pointer_traits<X::pointer>::rebind<const void>`
~~`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 difference between any two pointers in the allocation model.	`ptrdiff_t`
`typename X::template rebind<U>::other`	`Y`	For all `U` (including `T`), `Y::template rebind<T>::other` is `X`.	See Note A, below.
`*p`	`T&`		
`*q`	`T const&`	`*q` refers to the same object as `*p`	
`p->m`	type of `T::m`	*pre:* `(*p).m` is well-defined. equivalent to `(*p).m`	
`q->m`	type of `T::m`	*pre:* `(*q).m` is well-defined. equivalent to `(*q).m`	
`static_cast<X::pointer>(w)`	`X::pointer`	`static_cast<X::pointer>(w) == p`	
`static_cast<X::const_pointer>(z)`	`X::const_pointer`	`static_cast<X::const_pointer>(z) == q`	

`pointer d(nullptr);`
`pointer d = nullptr;`
`const_pointer e(nullptr);`
`const_pointer e = nullptr;`
		`d` and `e` are null pointers and need not be dereferenceable,
		static_cast<bool>`(d) == false`,
		static_cast<bool>`(e) == false`	

`void_pointer d(nullptr);`
`void_pointer d = nullptr;`
`const_void_pointer e(nullptr);`
`const_void_pointer e = nullptr;`
		`d` and `e` are null pointers and need not be dereferenceable,
		static_cast<bool>`(d) == false`,
		static_cast<bool>`(e) == false`	

`p`	contextually convertible to `bool`	false if `p` is a null pointer, else true	
`q`	contextually convertible to `bool`	false if `q` is a null pointer, else true	
`w`	contextually convertible to `bool`	false if `w` is a null pointer, else true	
`z`	contextually convertible	false if `z` is a null pointer, else true	

	to `bool`		
`!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`~~		
~~`a.address(s)`~~	~~`X::const_pointer`~~		
`a.allocate(n)` ~~`a.allocate(n,u)`~~	`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~~.[231] [*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 and shall not exit via an exception.	
`a1 != a2`	`bool`	same as `!(a1 == a2)`	
`a == b`	`bool`	same as `a == Y::rebind<T>::other(b)`	
`a != b`	`bool`	same as `!(a == b)`	
~~`X()`~~		~~creates a default instance. [*Note*: a destructor is assumed. — *end note*]~~	
`X a1(a);`		post: `a1 == a`. Shall not exit via an exception.	
`X a(b);`		post: `Y(a) == b, a == X(b)`. Shall not exit via an exception.	
`X a1(move(a));`		post: `a1` equals the prior value of `a`. Shall not exit via an exception.	
`X a(move(b));`		post: `a` equals the prior value of `X(b)`. Shall not exit via an exception.	
`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_contain`	`X`	Typically returns either `a` or `X()`	`return a;`

`er_copy_construction()`			
`X::propagate_on_container_copy_assignment`	identical-to or derived-from `true_type` or `false_type`	`true_type` if an allocator of type X should be copied when the client container is copy-assigned.	`false_type`
`X::propagate_on_container_move_assignment`	identical-to or derived-from `true_type` or `false_type`	`true_type` if an allocator of type X should be moved when the client container is move-assigned.	`false_type`
`X::propagate_on_container_swap`	identical-to or derived-from `true_type` or `false_type`	`true_type` if an allocator of type X should be swapped when the client container is swapped.	`false_type`

Note A: The member class `template rebind` in the table above is effectively a typedef template~~:~~. [*Note*: In general, if the name Allocator is bound to `SomeAllocator<T>`, then `Allocator::rebind<U>::other` is the same type as `SomeAllocator<U>`, where `SomeAllocator<T>::value_type` is `T` and `SomeAllocator<U>::value_type` is `U`. – *end note*] If `Allocator` is a class template instantiation of the form `SomeAllocator<T,` *args*`>`, where *args* is zero or more type arguments, and `Allocator` does not supply a `rebind` member template, the standard `allocator_traits` template uses `SomeAllocator<U,` *args*`>` in place of `Allocator::rebind<U>::other`, by default. For allocator types that are not template instantiations of the above form, no default is provided.

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 (20.2.1 [utility.arg.requirements]). No constructor, comparison operator, copy operation, move operation, or swap operation on these types shall exit via an exception. A default initialized object may have a singular value. A value-initialized object shall compare equal to `nullptr`. `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. Added the `void_pointer` and `const_void_pointer` types and the rules defining the minimal set of operations on pointer types.

2. Added default values, especially for the new features.

3. Changed the first argument to `construct` and `destroy` to be 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. Added the allocator propagation traits.

5. Removed the `address` function. The functionality of `address` is now provided by the `pointer_to` function in `pointer_traits`.

Note that there is no `select_on_container_move_construction()` function. After some consideration, we decided that a move construction operation for containers must run in 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.

A nothrow requirement was added to many allocator operations, including copy construction and equality comparison. It is painfully difficult to write a container correctly if the allocator can throw an exception on copying and comparison.

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

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

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

    SimpleAllocator(ctor args);

    template <class 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*]

The `uses_allocator` trait

Modify 20.8.1 [allocator.tag] as follows:

20.8.1 Allocator argument tag [allocator.tag]

```
namespace std {
  struct allocator_arg_t { };
  constexpr 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, 20.3.3~~tuple 20.5.2.1) have constructors with `allocator_arg_t` as the first argument, immediately followed by an argument of a type that satisfies the Allocator requirements (20.2.2).

Completely replace section 20.8.2 [allocator.traits] with the following [uses.allocator] section:

~~20.8.2 Allocator-related traits [allocator.traits]~~

~~**Etc.**~~

20.8.2 uses_allocator [uses.allocator]

20.8.2.1 uses_allocator_trait [uses.allocator.trait]

```
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). The implementation shall provide a definition that is derived from `true_type` if a type `T::allocator_type` exists and `is_convertible<Alloc, T::allocator_ type>::value != false`, otherwise it shall be derived from `false_type`. A program may specialize this ~~type~~template 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~~ can be constructed with an allocator where either:

- The first two arguments of a constructor have types `allocator_arg_t, Alloc`, or
- The last argument of a constructor has type `Alloc`.

~~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]

20.8.2.2 uses-allocator construction [uses.allocator.construction]

uses-allocator construction with allocator `Alloc` refers to the construction of an object, `obj` of type T, using constructor arguments `v1, v2, ..., vN` of types `V1, V2, …, VN` and an allocator `alloc` of type `Alloc` using the following rules:

- If `uses_allocator<T,Alloc>::value == false && is_constructible<T,V1, V2, …, VN>::value == true`, then `obj` is initialized as `obj(v1, v2, ..., vN).`
- Otherwise, if `(uses_allocator<T,Alloc>::value && is_constructible<T, allocator_arg_t, Alloc,V1, V2, …, VN>::value) == true`, then `obj` is initialized as `obj(allocator_arg, alloc, v1, v2, ..., vN).`
- Otherwise, if `(uses_allocator<T,Alloc>::value && is_constructible<T,V1, V2, …, VN, Alloc>::value) == true`, then `obj` is initialized as `obj(v1, v2, ..., vN, alloc).`
- Otherwise the request for uses-allocator construction is ill formed. [*Note*: an error will result if `uses_allocator<T,Alloc>::value` is true but the specific constructor does not take an allocator. This definition prevents a silent failure to pass the allocator to an element. – *end note*]

The `pointer_traits` template

Completely delete sections 20.8.3 [allocator.propagation] and 20.8.4 [allocator.element.concepts]:

~~20.8.3 Allocator propagation traits [allocator.propagation]~~

~~Etc.~~

~~20.8.4 Allocator-related element concepts [allocator.element.concepts]~~

~~Etc.~~

Insert new pointer traits section. (Note to the editor: You may want to move pointer_traits before uses_allocator):

20.8.3 Pointer traits [pointer.traits]

The struct template `pointer_traits` supplies a uniform interface to certain attributes of pointer-like types.

```
namespace std {
  template <class Ptr> struct pointer_traits
  {
    typedef Ptr      pointer;
    typedef see below element_type;
    typedef see below difference_type;

    template <class U> using rebind = see below;

    static pointer pointer_to(see below r);
```

```
  };

  template <class T>
  struct pointer_traits<T*>
  {
    typedef T         element_type;
    typedef T*        pointer;
    typedef ptrdiff_t difference_type;

    template <class U> using rebind = U*;

    static pointer pointer_to(see below r);
  };
}
```

20.8.3.1 Pointer traits member types

`typedef` *see below* `element_type;`

> *Type:* `Ptr::element_type` if such a type exists; otherwise, `T` if `Ptr` is a class template instantiation of the form `SomePointer<T,`*args*`>`, where *args* is zero or more type arguments; otherwise the specialization is ill-formed.

The technique for extracting the first template argument from an instantiation is borrowed from `boost::pointer_to_other`. A sample metafunction for this purpose follows:

```
    template <class _Tp> struct __first_param;

    template <template <class, class...> class _Tmplt, class _Tp,
              class... _Args>
    struct __first_param<_Tmplt<_Tp, _Args...> > {
        typedef _Tp type;
    };
```

`typedef` *see below* `difference_type;`

> *Type:* `Ptr::difference_type` if such a type exists; otherwise `std::ptrdiff_t`.

`template <class _U> using rebind =` *see below;*

> *Template Alias*; `Ptr::rebind<U>` if such a type exists; otherwise, if `Ptr` is a class template instantiation of the form `SomePointer<T,`*args*`>`, where *args* is zero or more type arguments; otherwise the instantiation of `rebind` is ill-formed.

The `boost::pointer_to_other` library provides an implementation of this default.

20.8.3.2 Pointer traits member functions

`Static pointer pointer_to(`*see below* `r);`

> *Preconditions:* For every object type `T`, `r` shall be T&; for *cv* `void`, the type of `r` is unspecified.
>
> *Returns:* a dereferenceable pointer to `r` obtained by calling `Ptr::pointer_to(r)`. An instantiation of this function is ill formed if `Ptr` does not have matching `pointer_to` static member function.

`Static pointer pointer_to(`*see below* `r);` (for the partial specialization, `pointer_traits<T*>`)

Preconditions: For every object type `T`, `r` shall be T&; for *cv* `void`, the type of `r` is unspecified.

Returns: `std::addressof(r).`

The `allocator_traits` template

Insert a new allocator traits section:

20.8.4 Allocator traits [allocator.traits]

The struct template `allocator_traits` supplies a uniform interface to all allocator types. A user specialization of `allocator_traits` in the `std` namespace for a specific set of allocators, `X` shall implement the interface described below in such a way that `allocator_traits<X>` meets the requirements of table 40 (see 20.2.2 [allocator.requirements]). An allocator cannot be a non-class type, however, even if `allocator_traits` supplies the entire required interface. [*Note:* thus, it is always possible to create a derived class from an allocator – *end note*]

Since most allocators are stateless, it is important support use of the empty-base-optimization by always allowing inheritance from allocator types.

```
namespace std {
  template <class 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;

    typedef see below propagate_on_container_copy_assignment;
    typedef see below propagate_on_container_move_assignment;
    typedef see below propagate_on_container_swap;

    template <class T> using rebind_alloc = see below;
    template <class 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 <class T, class... Args>
      static void construct(Alloc& a, T* p, Args&&... args);

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

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

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

20.8.3.1 Allocator traits member types

`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 `pointer_traits<pointer>::rebind<const value_type>`.

`typedef` *see below* `void_pointer;`

Type: `Alloc::void_pointer` if such a type exists, otherwise `pointer_traits<pointer>::rebind<void>`.

`typedef` *see below* `const_void_pointer;`

Type: `Alloc::const_void_pointer` if such a type exists, otherwise `pointer_traits<pointer>::rebind<const 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`.

`typedef` *see below* `propagate_on_container_copy_assignment;`

Type: `Alloc::propagate_on_container_copy_assignment` if such a type exists, otherwise `false_type`.

`typedef` *see below* `propagate_on_container_move_assignment;`

Type: `Alloc::propagate_on_container_move_assignment` if such a type exists, otherwise `false_type`.

`typedef` *see below* `propagate_on_container_swap;`

Type: `Alloc::propagate_on_container_swap` if such a type exists, otherwise `false_type`.

`template <class T> using rebind_alloc =` *see below*`;`

Template Alias; `Alloc::rebind<U>::other` if such a type exists; otherwise, if `Alloc` is a class template instantiation of the form `Alloc<T,`*args*`>`, where *args* is zero or more type arguments; otherwise the instantiation of `rebind_alloc` is ill-formed.

20.8.3.2 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 <class T, class... 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 <class 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 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`.

Changes to the default allocator

Modify the declarations of `construct()` and `destroy()` in section 20.8.5 [default.allocator] as follows:

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

Also the description of `construct()` and `destroy()` in section 20.8.5.1 [allocator.members]:

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

Effects: `::new((void *)p) `~~`T`~~`U(std::forward<Args>(args)...)`

`template<class U> void destroy(`~~`pointer`~~`U* p);`

Effects: `p->~`~~`T`~~`U()`

Completely delete section 20.8.9 [construct.element]:

~~**20.8.9 construct_element [construct.element]**~~

~~Etc.~~

Scoped allocator adaptors

Completely replace section 20.8.6 [allocator.adaptor] with the following:

20.8.6 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 <class OuterAlloc, class... 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;

    typedef see below propagate_on_container_copy_assignment;
    typedef see below propagate_on_container_move_assignment;
    typedef see below propagate_on_container_swap;

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

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

    scoped_allocator_adaptor(const scoped_allocator_adaptor& other);

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

```
    template <class 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 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 <class T, class... 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 <class T>
      void destroy(T* p);

    // Allocator propagation functions.
    scoped_allocator_adaptor select_on_container_copy_construction() const;
  };

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

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

20.8.6.1 scoped_allocator_adaptor member types [allocator.adaptor.types]

`typedef` *see below* `inner_allocator_type;`

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

`typedef` *see below* `propagate_on_container_copy_assignment;`

Type: `true_type` if `allocator_traits<A>::propagate_on_container_copy_assignment::value` is true for any `A` in the set of `OuterAlloc` and `InnerAllocs...`; otherwise `false_type`.

`typedef` *see below* `propagate_on_container_move_assignment;`

Type: `true_type` if `allocator_traits<A>::propagate_on_container_move_assignment::value` is true for any `A` in the set of `OuterAlloc` and `InnerAllocs...`; otherwise `false_type`.

`typedef` *see below* `propagate_on_container_swap;`

Type: `true_type` if `allocator_traits<A>::propagate_on_container_swap::value` is true for any `A` in the set of `OuterAlloc` and `InnerAllocs...`; otherwise `false_type`.

20.8.6.2 scoped_allocator_adaptor constructors [allocator.adaptor.cntr]

```
scoped_allocator_adaptor();
```

Effects: value-initializes the `OuterAlloc` base class and the `inner` allocator object

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

Requires: `OuterAlloc` shall be constructible from OuterA2.

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 do not require perfect forwarding of innerAllocs because it is not in a deduced context and cannot, therefore, would require significant meta-programming to ensure perfect forwarding. Perfect forwarding of allocators is rarely a huge win because allocators should be efficiently copiable.

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

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

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

Requires: `OuterAlloc` shall be constructible from OuterA2.

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

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

Requires: `OuterAlloc` shall be constructible from OuterA2.

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

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

Effects: let *`OUTERMOST(x)`* be `x` if `x` does not have an `outer_allocator()` function, and *`OUTERMOST(x.outer_allocator())`* otherwise.

- If `uses_allocator<T,inner_allocator_type>::value == false && is_constructible<T,Args...>::value == true`, call *OUTERMOST(*`*this`*)*`.construct(p, std::forward<Args>(args)...)`.

- Otherwise, if (`uses_allocator<T,inner_allocator_type>::value && is_constructible<T,allocator_arg_t,inner_allocator_type,Args...>::value) == true`, then calls *OUTERMOST(*`*this`*)*`.construct(p, allocator_arg, inner_allocator(), std::forward<Args>(args)...)`.

- Otherwise, if (`uses_allocator<T,inner_allocator_type>::value && is_constructible<T,Args...,inner_allocator_type>::value) == true`, then calls *OUTERMOST(*`*this`*)*`.construct(p,std::forward<Args>(args)..., inner_allocator())`.

- Otherwise the program1 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 <class T>
  void destroy(T* p);
```

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

```
scoped_allocator_adaptor select_on_container_copy_construction() const;
```

Returns: a new `scoped_allocator_adaptor` where each allocator, `A`, in the adaptor is initialized from the result of calling `allocator_traits<A>::select_on_container_copy_construction` on the corresponding allocator in `*this`.

Changes to container and string wording

In all container and string classes and match_results, and sub_match, make the following replacements in the `typedef` sections of the class template definitions:

Old type	New type
`Allocator::value_type`	`allocator_traits<value_type>::value_type`
`Allocator::reference`	`value_type&`
`Allocator::const_reference`	`const value_type&`
`Allocator::pointer`	`allocator_traits<Allocator>::pointer`
`Allocator::const_pointer`	`allocator_traits<Allocator>::const_pointer`
`Allocator::difference_type`	`allocator_traits<Allocator>::difference_type`
`Allocator::size_type`	`allocator_traits<Allocator>::size_type`

Add a phrase to Section 21.4 [basic.string] paragraph 3:

The member functions of `basic_string` use an object of the `Allocator` class passed as a template parameter to allocate and free storage for the contained char-like objects. [235]

The class template `basic_string` conforms to the requirements for a Sequence Container (23.2.3), for a Reversible Container (23.2), and for an Allocator-aware container (93), except that `basic_string` does not construct or destroy its elements using `allocator_traits<Alloc>::construct` and `allocator_traits<Alloc>::destroy`. The iterators supported by `basic_string` are random access iterators (24.2.5).

Modify paragraph 21.4.1/2 [string.require] as follows:

In every specialization `basic_string<charT, traits, Allocator>`, the nested type `allocator_traits<Allocator>::value_type` shall name the same type as `charT`. Every object of type `basic_string<charT, traits, Allocator>` shall use an object of type `Allocator` to allocate and free storage for the contained `charT` objects as needed. The Allocator object used shall be obtained as described in Section 23.2.1 [container.requirements.general]~~a copy of the Allocator object passed to the basic_string object's constructor or, if the constructor does not take an Allocator argument, a copy of a default-constructed Allocator object~~.

Modify paragraph 21.4.2/2 [string.cons]:

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

Effects: Constructs an object of class basic_string as indicated in Table 60.~~ In the first form, the stored Allocator value is copied from str.get_allocator().~~ In the second form,~~ the stored Allocator value is move constructed from str.get_allocator(), and~~ str is left in a valid state with an unspecified value.

Change footnote 237 in 21.4.6.1:

237) `reserve()` uses `allocator_traits<Allocator>::allocate()` which may throw an appropriate exception.

Change 21.4.7.1/3 [string.accessor]

`allocator_type get_allocator() const;`

Returns: a copy of the `Allocator` object used to construct the string or, if that allocator has been replaced, a copy of the most recent replacement.

Change section 23.2.1 [container.requirements.general], paragraphs 3 and 4 as follows:

3 For the components affected by this clause that declare an `allocator_type`, o~~O~~bjects stored in these components shall be constructed using ~~construct_element (20.8.9)~~the `allocator_traits<allocator_type>::construct` function and destroyed using the `allocator_traits<allocator_type>::destroy` function (20.8.3.2 [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 and 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 90, 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 90:

~~Table 90 — ConstructibleAsElement<A, T, Args> requirements [constructibleaselement]~~

~~Etc.~~

Modify the notes after Table 91 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.~~

Modify Section 23.2.1 [container.requirements.general], paragraph 9:

Unless otherwise specified, all containers defined in this clause obtain memory using an allocator (See 20.2.2). Copy ~~and move~~ constructors for these container types obtain an allocator by calling ~~Allocator<allocator_type>::select_for_copy_construction or Allocator<allocator_type>::select_for_move_construction~~ `allocator_traits<allocator_type>::select_on_container_copy_construction` on their ~~respective~~ first parameters. Move constructors obtain an allocator by move-construction of the allocator belonging to the container being moved. Such move-construction of the allocator shall not exit with an exception. All other constructors for these container types take an `Allocator` argument (20.2.2), an allocator whose value type is the same as the container's value type. [*Note:* If an invocation of a constructor uses the default value of an optional allocator argument, then the `Allocator` type must support value initialization – *end note*] 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<allocator_type>::do_on_container_copy_assignment, Allocator<allocator_type>::do_on_container_copy_assignment, Allocator<allocator_type>::do_on_swap~~ copy assignment, move assignment, or swapping of the allocator only if `allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value`, `allocator_traits<allocator_type>::propagate_on_container_move_assignment::value`, or `allocator_traits<allocator_type>::propagate_on_container_swap::value` is true within the implementation of the corresponding container operation. A call to a container's `swap` function will yield undefined behavior unless the objects being swapped have allocators that compare equal or `allocator_traits<allocator_type>::propagate_on_container_swap::value` is true. ~~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, if that allocator has been replaced, a copy of the most recent replacement.

In table 93 (Allocator-aware container requirements), modify selected rows as shown:

`X(t,m)` `X u(t,m);`	~~Requires: ConstructibleAsElement<A, T, T>~~ post: `u == t`, `get_allocator() == m`	linear
`X(rv)` `X u(rv);`	requires: move-construction of A shall not exit with an exception. post: `u` shall have the same elements as `rv` had before this construction; `get_allocator()` shall be the same as the value of `rv.get_allocator()` before this construction.	constant
`X(rv,m)` `X u(rv,m);`	~~Requires: ConstructibleAsElement<A, T, T&&>~~ post: `u` shall ~~be equal to the value~~have the same elements, or copies of the elements, that `rv` had before this construction; `get_allocator() == m`	constant if `m == rv.get_allocator()`, otherwise linear

Rename `construct_element` to `construct` in section 23.3.7 [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 ~~AllocatableElement::construct (23.2) (20.8.4)~~ `allocator_traits::construct` (20.8.4.2) is not used to construct these values.

Tuple changes

In section 20.5.1, [tuple.general], remove the specialization `constructible_with_allocator_prefix`:

~~`template <class... Types>`~~
~~`  struct constructible_with_allocator_prefix<tuple<Types...> >;`~~

Same for 20.5.2.8 [tuple.traits]:

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

> ~~[Note: Specialization of this trait informs other library components that tuple can be constructed with an allocator prefix argument. —end note]~~

Change section 2.5.2.1 [tuple.], paragraphs 21 and 22 as follows:

```
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, const 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~~meet the requirements for an Allocator (20.2.2).
>
> *Effects*: Equivalent to the preceding constructors except that each element is constructed with *uses-allocator construction* with allocator `a` (20.8.2.2 [uses.allocator.construction]). ~~the allocator argument is passed conditionally to the constructor of each element~~. ~~Each member is allocator constructed (20.8.2) with a.~~

The term *allocator constructed* was removed from section 20.8.2, and thus had to be described in place.

Function changes

The WP also has the definition for a function function::assign(F&, A), where A is an allocator. This function and its definition (function(f,a).swap(*this)) make no sense to me. Should it be removed?

In section 20.7.15.2 [func.wrap.func], remove the obsolete trait:

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

Modify section 20.7.15.2.1 [func.wrap.func.con], as follows

20.7.15.2.1 function construct/copy/destroy [func.wrap.func.con]

For all constructors in this clause that have arguments lists beginning with types `allocator_arg_t, const A&`, type `A` shall be assumed by the implementation to conform to the allocator requirements in Table 40 [allocator.requirements]. A copy of the allocator argument is used to allocate memory, if necessary for the internal data structures of the constructed `function` object.

```
explicit function();
template <class A> function(allocator_arg_t, const A& a);
```

Postconditions: !*this.

Throws: nothing.

```
function(nullptr_t);
template <class A> function(allocator_arg_t, const A& a, nullptr_t);
```

Postconditions: !*this.

Throws: nothing.

```
function(const function& f);
template <class A> function(allocator_arg_t, const A& a, const function& f);
```

Postconditions: !*this if !f; otherwise, *this targets a copy of f.target().

Throws: shall not throw exceptions if f's target is a function pointer or a function object passed via `reference_wrapper`. Otherwise, may throw bad_alloc or any exception thrown by the copy constructor of the stored function object. [*Note*: Implementations are encouraged to avoid the use of dynamically allocated memory for small function objects, e.g., where f's target is an object holding only a pointer or reference to an object and a member function pointer. —*end note*]

```
function(function&& f);
template <class A> function(allocator_arg_t, const A& a, function&& f);
```

Effects: If !f, *this has no target; otherwise, move-constructs the target of f into the target of *this, leaving f in a valid state with an unspecified value.

`template<class F> function(F f);`
`template<class F, class A> function(allocator_arg_t, const A& a, F f);`
~~`template<class F, class A> function(F f, const A& a);`~~

Requires: f shall be callable for argument types ArgTypes and return type R. The copy constructor and destructor of A shall not throw exceptions.

Postconditions: !*this if any of the following hold:

— f is a NULL function pointer.

> — f is a NULL member function pointer.
>
> — F is an instance of the function class template, and !f
>
> Otherwise, *this targets a copy of f or std::move(f) if f is not a pointer to member function, and targets a copy of mem_fn(f) if f is a pointer to member function. [*Note*: implementations are encouraged to avoid the use of dynamically allocated memory for small function objects, for example, where f's target is an object holding only a pointer or reference to an object and a member function pointer. —*end note*]
>
> *Throws*: shall not throw exceptions when f is a function pointer or a `reference_wrapper<T>` for some T. Otherwise, may throw `bad_alloc` or any exception thrown by F's copy or move constructor.

A definition of the allocator constructors for `function` was completely absent from the current WP and previous drafts.

Changes to `match_results`

Change 28.10/2 [re.results]:

> The class template `match_results` shall satisfy the requirements of an allocator-aware container and of a sequence container, as specified in 23.2.3, except that only operations defined for const-qualified sequence containers are supported.

Change 28.10.5/1 [re.results.all]:

> **28.10.5 match_results allocator [re.results.all]**
>
> `allocator_type get_allocator() const;`
>
> *Effects*: Returns a copy of the Allocator that was passed to the object's constructor or, if that allocator has been replaced, a copy of the most recent replacement.

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 `allocator_traits<Alloc>::void_pointer` to the list of types on which an iterator may depend.
2. Add `allocator_traits<Alloc>::void_pointer` and `allocator_traits<Alloc>::const_void_pointer` to the list of types on which a const_iterator may depend.

Acknowledgements

Special thanks to Howard Hinnant for his pointer_traits idea and Daniel Krügler for his thorough review. Thanks to John Lakos, Alisdair Merideth, Ion Gaztañaga, 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
N2981: Proposal to Simplify pair (Rev 3)
N2945: Proposal to Simplify pair (Rev 2)