

Doc No: N2654=08-0164
Date: 2008-06-27
Author: Pablo Halpern
Bloomberg, L.P.

phalpern@halpernwrightsoftware.com

Allocator Concepts (revision 1)

Contents

Changes from N2641	1
Summary	2
Document Conventions	3
Proposed Wording.....	3
The addressof Function.....	3
Header changes	3
Allocator Concepts.....	7
Allocator-related Element Concepts	13
Allocator Propagation Concepts.....	18
Scoped Allocator Adaptor	21
Element construction.....	24
Acknowledgements	25
References	25

Changes from N2641

Most of the changes in this document are minor technical fixes:

- A few typos.
- Add missing template `<ObjectType T>` before declaration of `addressof`.
- Add missing concept parameter lists to `Allocator`, `RandomAccessAllocator`, `SimpleAllocator`, and `MinimalAllocator`.
- Correct syntax of `ScopedAllocator` concept to correctly refine `Allocator`.
- Use `ObjectType` in several places where `T` must be an object type.
- Changed `Constructible` to `HasConstructor` as per core library concepts proposal. Also added `HasConstructor` requirement in a few missing places.

- The `ConstructibleWithAllocatorSuffix` and `ConstructibleWithAllocatorPrefix` concepts are now mutually-exclusive, with the latter preferred for types that match both.
- Add missing “&&” in a few places where a variadic template parameter pack was being forwarded to a concept in a `requires` clause.

Summary

This paper defines concepts to replace the allocator-related type traits defined in N2554 (*The Scoped Allocator Model*) and N2525 (*Allocator-specific Move and Swap*) and specifies concept constraints and concept maps for a number of library classes and class templates. Most of these concepts are used in N2623: *Concepts for the C++0x Standard Library: Containers*. Other parts of the library that are affected by these concepts are described here.

There are a number of notable consequences of adding concepts to allocators:

1. The pointer types in the `Allocator` concept is now defined with sufficient precision that we are able to remove the weasel words that previously prevented portable use of fancy pointer types in allocators.
2. Several traits defined in the WP can be replaced by auto concepts, freeing the programmer from having to specify them explicitly for his/her types.
3. Not all allocators require random-access pointer types. Those containers that require random-access would use the new `RandomAccessAllocator` refinement of the `Allocator` concept.
4. User-defined container types that don't want to deal with non-raw pointer types, etc., can be constrained with the `SimpleAllocator` concept, which requires that the allocator use raw pointers.
5. An allocator author can allow a container to bypass calls to `construct()` and `destroy()` by specifying that the allocator models the `MinimalAllocator` concept. This is useful for, e.g., `vector<int>`, where no destructor is needed and where `resize` can be specialized to use `memset()`. The default allocator is a `MinimalAllocator`.
6. A default implementation of `Allocator<X>::construct()` allows C++03 allocators to work as C++0x allocators by automatically providing a variadic `construct()` function.

Document Conventions

All section names and numbers are relative to the March 2008 working draft, N2588.

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 changes resulting from such guidance will be minor and will not delay acceptance of this proposal in the same meeting at which it is presented.

Proposed Wording

The addressof Function

In section 20.6 [memory] within the synopsis of header <memory>, add before `uninitialized_copy`:

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

In section 20.6.10, insert the following:

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

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

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.

I'm not sure both overloads of `addressof` are necessary or if the rvalue form is sufficient to bind to all references.

Header changes

Insert the following at the top of section 20.6:

Header <memory_concepts> synopsis:

```
namespace std {  
    // Allocator concepts  
    auto concept Allocator<typename Alloc> see below  
    auto concept RandomAccessAllocator<typename Alloc> see below
```

```

auto concept SimpleAllocator<typename Alloc> see below
concept MinimalAllocator<typename Alloc> see below

// Scoped allocators
concept ScopedAllocator<typename Alloc> see below

// Allocator-related element concepts
auto concept UsesAllocator<class T, class Alloc> see below

auto concept ConstructibleWithAllocatorSuffix<class T, class Alloc,
                                             class... Args> see below

auto concept ConstructibleWithAllocatorPrefix<class T, class Alloc,
                                             class... Args> see below

concept ConstructibleWithAllocator<class T, class Alloc,
                                   class... Args> see below
template <Allocator Alloc, class T, class... Args>
  requires !UsesAllocator<T, Alloc>
         && HasConstructor<T, Args&&...>
  concept_map ConstructibleWithAllocator<Alloc, T, Args&&...> see
below
template <Allocator Alloc, class T, class... Args>
  requires ConstructibleWithAllocatorPrefix<T, Alloc, Args&&...>
  concept_map ConstructibleWithAllocator<Alloc, T, Args&&...> see
below
template <Allocator Alloc, class T, class... Args>
  requires ConstructibleWithAllocatorSuffix<T, Alloc, Args&&...>
  concept_map ConstructibleWithAllocator<Alloc, T, Args&&...> see
below

concept ConstructibleAsElement<class Alloc, class T,
                              class... Args> see below

template <Allocator Alloc, class T, class... Args>
  requires !ScopedAllocator<Alloc>
         && HasConstructor<T, Args&&...>
  concept_map ConstructibleAsElement<Alloc, T, Args&&...> see below
template <ScopedAllocator Alloc, class T, class... Args>
  requires !UsesAllocator<T, Alloc::inner_allocator >
         && HasConstructor<T, Args&&...>
  concept_map ConstructibleAsElement<Alloc, T, Args&&...> see below
template <ScopedAllocator Alloc, class T, class... Args>
  requires ConstructibleWithAllocatorPrefix<T,
                                           Alloc::inner_allocator, Args&&...>
  concept_map ConstructibleAsElement<Alloc, T, Args&&...> see below
template <ScopedAllocator Alloc, class T, class... Args>
  requires ConstructibleWithAllocatorSuffix<T,

```

```

        Alloc::inner_allocator, Args&&...>
    concept_map ConstructibleAsElement<Alloc,T,Args&&...> see below

// Allocator-propagation concepts
concept AllocatorPropagation<typename Alloc> see below
concept AllocatorPropagateNever<typename Alloc> see below
concept AllocatorPropagateOnCopyConstruction<typename Alloc> see
below
concept AllocatorPropagateOnMoveAssignment<typename Alloc> see below
concept AllocatorPropagateOnCopyAssignment<typename Alloc> see below

template <Allocator Alloc>
    requires ! AllocatorPropagateNever<Alloc> &&
             ! AllocatorPropagateOnMoveAssignment<Alloc> &&
             ! AllocatorPropagateOnCopyAssignment<Alloc>
    concept_map AllocatorPropagateOnCopyConstruction<Alloc> {
}
}

```

In section 20.6, header <memory> synopsis, remove declarations of allocator-related traits:

```

// 20.6.2, allocator-related traits
template <class T, class Alloc> struct uses_allocator;
template <class Alloc> struct is_scoped_allocator;
template <class T> struct constructible_with_allocator_suffix;
template <class T> struct constructible_with_allocator_prefix;
// 20.6.3, 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;

```

Also concept maps and allocator-related constraints:

```

// 20.6.5, the default allocator:
template <class T> class allocator;
template <ObjectType T>
    concept map MinimalAllocator<allocator<T> > { };
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();

// 20.6.6, scoped allocator adaptor
template <classAllocator OuterA, classAllocator InnerA = void>

```

```

class scoped_allocator_adaptor;
template <classAllocator Alloc>
class scoped_allocator_adaptor<Alloc, void>;
template <classAllocator OuterA, classAllocator InnerA>
struct is_scoped_allocator<scoped_allocator_adaptor<OuterA,
InnerA>>
 : true_type { };
requires ObjectType<OuterA::value type>
concept map ScopedAllocator<
scoped_allocator_adaptor<OuterA, InnerA> > { }
template <classAllocator OuterA, classAllocator InnerA>
struct allocator_propagate_never<scoped_allocator_adaptor<OuterA,
InnerA>>
 : true_type { };
requires ObjectType<OuterA::value type>
concept map AllocatorPropagateNever<
scoped_allocator_adaptor<OuterA, InnerA> > { }
template<typenameAllocator OuterA1, typenameAllocator OuterA2,
typenameAllocator InnerA>
bool operator==(const scoped_allocator_adaptor<OuterA1,InnerA>& a,
const scoped_allocator_adaptor<OuterA2,InnerA>& b);
template<typenameAllocator OuterA1, typenameAllocator OuterA2,
typenameAllocator InnerA>
bool operator!=(const scoped_allocator_adaptor<OuterA1,InnerA>& a,
const scoped_allocator_adaptor<OuterA2,InnerA>& b);

// 20.6.7, raw storage iterator:
template <class OutputIterator, class T> class raw_storage_iterator;

// 20.6.8, temporary buffers:
template <class T>
pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
template <class T>
void return_temporary_buffer(T* p);

// 20.6.9, construct element
template <class Alloc, class T, class... Args>
void construct_element(Alloc& alloc, T& r, Args&&... args);
template <Allocator Alloc, class T, class... Args>
requires !ScopedAllocator<Alloc>
&& HasConstructor<T, Args&&...>
void construct_element(Alloc& a, T& r, Args&&... args);
template <ScopedAllocator Alloc, class T, class... Args>
requires !UsesAllocator<T, Alloc::inner_allocator>
&& HasConstructor<T, Args&&...>
void construct_element(Alloc& a, T& r, Args&&... args);
template <ScopedAllocator Alloc, class T, class... Args>
requires ConstructibleWithAllocatorPrefix<T,
Alloc::inner_allocator, Args&&...>
void construct_element(Alloc& a, T& r, Args&&... args);
template <ScopedAllocator Alloc, class T, class... Args>

```

```
requires ConstructibleWithAllocatorSuffix<T,  
        Alloc::inner_allocator, Args&&...>  
    void construct_element(Alloc& a, T& r, Args&&... args);
```

Allocator Concepts

Remove section 20.1.2 [allocator.requirements] entirely.

Allocator concepts have been consolidated into section 20.6.

Insert the following section before the current section 2.6.2:

We have kept most of the text of [allocator.requirements] here, although much of it has been moved from tables into numbered paragraphs when translating the allocator requirements into concepts. Text that was copied almost verbatim from [allocator.requirements] is shown with appropriate mark-up.

20.6.2 Allocators [allocator.introduction]

The library describes a standard set of requirements for allocators, which are 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 \(clause 23\)](#) are parameterized in terms of allocators.

~~Table 39 describes the requirements on types manipulated through allocators. The Allocator concept describes the requirements on allocators. The RandomAccessAllocator, SimpleAllocator, and MinimalAllocator concepts refine Allocator and specify additional constraints required by certain containers. All the operations on the allocators are expected to be amortized constant time. Each allocator operation shall have amortized constant time complexity. Table 40 describes the requirements on allocator types.~~

The above are modified versions of the [allocator.requirements], paragraphs 1 and 2.

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 above is a verbatim copy of [allocator.requirements], paragraph 6.

Note that Tables 39 and 40 are gone. Also gone are the weasel words preventing portable use of allocators with non-raw pointer types ([allocator.requirements], paragraphs 4 and 5). A moment of silence please!

20.6.2.1 Allocator Concept [allocator.concept]

```
auto concept Allocator<typename X> :  
    CopyConstructible<X>, EqualityComparable<X> {
```

```

ObjectType value_type = typename X::value_type;
Dereferenceable pointer = see below;
Dereferenceable const_pointer = see below;
typename generic_pointer = void*;
typename const_generic_pointer = const void*;
typename reference = value_type&;
typename const_reference = const value_type&;
UnsignedIntegralLike size_type = see below;
template<ObjectType T> class rebind = see below;

requires Destructible<value_type>;
requires Convertible<pointer, const_pointer> &&
    Convertible<pointer, generic_pointer> &&
    SameType<pointer::reference, value_type&> &&
    SameType<pointer::reference, reference>;
requires Convertible<const_pointer, const_generic_pointer> &&
    SameType<const_pointer::reference, const value_type&> &&
    SameType<const_pointer::reference, const_reference>;
requires SameType<rebind<value_type>, X>;
requires SameType<generic_pointer,
    rebind<unspecified unique type>::generic_pointer>;
    // see description of generic_pointer, below
requires SameType<const_generic_pointer,
    rebind<unspecified unique type>::const_generic_pointer>;
    // see description of generic_pointer, below

pointer X::allocate(size_type n);
pointer X::allocate(size_type n, const_generic_pointer p);
void X::deallocate(pointer p, size_type n);
size_type X::max_size() const {
    return numeric_limits<size_type>::max(); }

template<ObjectType T>
    X::X(const rebind<T>& y);

template<typename... Args>
    requires HasConstructor<value_type, Args&&...>
    void X::construct(pointer p, Args&&... args)
    {
        ::new ((void*) addressof(*p))
            value_type(forward<Args>(args)...);
    }

void X::destroy(pointer p) {
    addressof(*p)->~value_type();
}

pointer X::address(reference r) const {
    return addressof(r); // see below
}

```



```

    const_pointer X::address(const_reference r) const {
        return addressof(r); // see below
    }
}

```

```
ObjectType value_type;
```

Type: The type of object allocated by X.

```
Dereferenceable pointer;
Dereferenceable const_pointer;
```

Type: A pointer-like (const pointer-like) type used to refer to memory allocated by objects of type X. The default pointer type is X::pointer if such a type is declared and value_type* otherwise. The default const_pointer type is X::const_pointer if such a type is declared and const value* otherwise.

Defining the default type this way allows the programmer to define an allocator without specifying the pointer type, in the common case where the pointer type is simply value_type*. A conditional-default type within a concept can be implemented by refining special “base concepts” with appropriate constraints. The names and contents of these base concepts is an implementation detail and is not part of the standard.

```
typename generic_pointer;
typename const_generic_pointer;
```

A type that can store value of a pointer (const_pointer) from any allocator in the same family as X and which will produce the same value when explicitly converted back to that pointer type. For any two allocators X, and Y of the same family, the implementation of a library facility using Allocator<X> and Allocator<Y>, is permitted to add additional requirements, SameType<Allocator<X>::generic_pointer, Allocator<Y>::generic_pointer> and SameType<Allocator<X>::const_generic_pointer, Allocator<Y>::const_generic_pointer>

[*Example:*

```

template<ObjectType T, Allocator Alloc = allocator<T> >
    requires Destructible<T> &&
    SameType<Alloc::generic_pointer,
            Alloc::Rebind<list_node<T>>::generic_pointer> &&
    SameType<Alloc::const_generic_pointer,
            Alloc::Rebind<list_node<T>>::const_generic_pointer>
    class list;

```

end example]

The addition of generic_pointer eliminates the common trick of using rebind<void>::other::pointer as a way to represent a pointer of unknown type. The trick was never actually sufficient, as there was never a requirement that pointer

be convertible to/from a `rebind<void>::pointer` or vice-versa. In addition, `rebind<void>::other` requires that the allocator be specialized for `void`. By introducing `generic_pointer` and formalizing the convertibility requirements, we eliminate the need for `void` specializations and for the `AllocatorGenerator` concept proposed in an earlier version of the core library concepts proposal.

There is no way, using concepts, to indicate that that all of the rebound allocator's `generic_pointer_types` must be the same. We must, therefore, allow a library facility to require that a specific set of rebound allocator's `generic_pointer_types` must be the same.

```
typename reference;  
typename const_reference;
```

A reference (const reference) to a `value_type` object.

We make no attempt to allow for “smart references” in allocators.

```
UnsignedIntegralLike size_type;
```

Type: a type that can represent the size of the largest object in the allocation model. The default `size_type` is `X::size_type` if such a type is declared and `std::size_t` otherwise.

```
template<ObjectType T> class rebind;
```

Class Template: The associated template `rebind` is a template that produces allocators in the same family as `X`: if the name `X` is bound to `SomeAllocator<value_type>`, then `rebind<U>` is the same type as `SomeAllocator<U>`. The resulting type `SomeAllocator<U>` shall meet the requirements of the `Allocator` concept. The default value for `rebind` is a template `R` for which `R<U>` is `X::template rebind<U>::other`.

The aforementioned default value for `rebind` can be implemented as follows:

```
template<typename Alloc> struct rebind_allocator {  
    template<typename U>  
        using rebind = typename Alloc::template rebind<U>::other;  
};
```

The default value for `rebind` in the `Allocator` concept is, therefore, `rebind_allocator<X>::template rebind`.

```
pointer X::allocate(size_type n);  
pointer X::allocate(size_type n, const_generic_pointer hint);
```

Effects: Memory is allocated for `n` objects of type `value_type` but the objects are not constructed. [*Footnote:* It is intended that `a.allocate` be an efficient means of allocating a single

object of type T, even when sizeof(T) is small. That is, there is no need for a container to maintain its own “free list”. – *end footnote*] The optional argument, p, may

Returns: A pointer to the allocated memory. [*Note:* If n == 0, the return value is unspecified. If n > 1, the means by which a program gains access to the second and subsequent allocated objects is determined outside of the Allocator concept. See RandomAccessAllocator, below, for one common approach. – *end note*]

Throws: allocate may raise an appropriate exception.

Remark: The use of hint is unspecified, but intended as an aid to locality if an implementation so desires. [*Note:* In a container member function, the address of an adjacent element is often a good choice to pass for the hint argument. — *end note*]

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

Preconditions: All n value_type 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. [*Note:* p shall not be singular. — *end note*]

Throws: Does not throw exceptions.

```
size_type X::max_size();
```

Returns: the largest value that can meaningfully be passed to X::allocate()

```
template<typename... Args>  
requires HasConstructor<value_type, Args&&...>  
void X::construct(pointer p, Args&&... args);
```

Effects: Calls the constructor for the object at p, using the args constructor arguments.

Default: ::new ((void*) addressof(*p)) value_type(forward<Args>(args)...);

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

Effects: Calls the destructor on the object at p but does not deallocate it.

Default: addressof(*p) ->~value_type();

```
pointer X::address(reference r) const;  
const_pointer X::address(const_reference r) const;
```

Precondition: r is a reference to an object that was allocated from a copy of this allocator.

Returns: a pointer to the object referred-to by r. This concept defines a default implementation of address only if pointer is the same as value_type*.

2.1.2.2 Random-access Allocators [concept.random.allocator]

The `RandomAccessAllocator` concept is a refinement of `Allocator` that requires that the `pointer` and `const_pointer` types be random-access iterators, i.e., that they provide random access to the items in an array allocated from the allocator.

```
auto concept RandomAccessAllocator<typename X> : Allocator<X> {
    requires MutableRandomAccessIterator<pointer>;
    requires RandomAccessIterator<const_pointer>;
    SignedIntegralLike difference_type =
        MutableRandomAccessIterator<pointer>::difference_type;
}
SignedIntegralLike difference_type;
```

Type: a type that can represent the difference between any two pointers in the allocation model.

2.1.2.3 Simple Allocators [concept.simple.allocator]

The `SimpleAllocator` concept is a refinement of `RandomAccessAllocator` that requires that the `pointer` and `const_pointer` types be raw pointers to `value_type`.

```
auto concept SimpleAllocator<typename X> : RandomAccessAllocator<X>
{
    requires SameType<X::pointer, X::value_type*>;
    requires SameType<X::const_pointer, const X::value_type*>;
    requires SameType<X::generic_pointer, void*>;
    requires SameType<X::const_generic_pointer, const void*>;
}
```

2.1.2.4 Minimal Allocators [concept.minimal.allocator]

The `MinimalAllocator` concept is a refinement of `SimpleAllocator`. If a `MinimalAllocator<X> concept_map` exists for a specific allocator type `X`, it indicates that a program using `X` may directly call constructors and destructors of `value_type` without calling `X::construct` or `X::destruct`.

```
concept MinimalAllocator<typename X> : SimpleAllocator<X> {
}
```

2.1.2.4 Scoped Allocators [concept.minimal.allocator]

The `ScopedAllocator` concept is a refinement of `Allocator`. If a `ScopedAllocator<X>` concept map exists for a specific allocator type `X`, it indicates that `X` is a *Scoped Allocator*. A scoped allocator specifies the memory resource to be used by a container (as any other allocator does) and also specifies an *inner allocator* resource to be used by every element in the container. A `ScopedAllocator` cannot be a `MinimalAllocator`.

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

Allocator-related Element Concepts

Replace section 20.6.2 (*Allocator-related Traits*) with the following section:

~~2.6.2 Allocator-related traits [allocator.traits]~~

20.6.3 Allocator-related Element Concepts [allocator.element.concepts]

Replace the `uses_allocator` trait with the `UsesAllocator` concept:

```
template <class T, class Alloc> struct uses_allocator; see below  
auto concept UsesAllocator<typename T, typename Alloc> {  
    requires Allocator<Alloc>;  
    typename allocator_type = T::allocator_type;  
    requires Convertible<Alloc, allocator_type>;  
}
```

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

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

Remove [allocator.traits], paragraph 3:

~~The class templates, `is_scoped_allocator`, `constructible_with_allocator_suffix`, and `constructible_with_allocator_prefix` meet the UnaryTypeTrait requirements ([meta.reqmts] 20.4.1). Each of these templates shall be publicly derived directly or indirectly from true_type if the corresponding condition is true, otherwise from false_type. All are elective traits; they are not computed automatically by determining an intrinsic quality of the type, but rather indicate a deliberate choice by the author of the type. A program may specialize these traits for user-defined types provided that the user-defined type meets the requirement of the trait. However, a program is never required to specialize these traits.~~

Remove the definition of `is_scoped_allocator`. The definition of the `ScopedAllocator` concept was added to the previous section.

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

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

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

Replace the `constructible_with_allocator_suffix/prefix` traits with concepts:

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

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

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

{Example:

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

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

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

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

—end-example}
```

```
auto concept
ConstructibleWithAllocatorSuffix<class T, class Alloc,
class... Args>
: UsesAllocator<T, Alloc>
{
requires ! ConstructibleWithAllocatorPrefix<T, Alloc, Args&&...>
requires HasConstructor<T, Args&&..., allocator type>;
}
```

Remark: an (automatically generated) concept map for a given set of parameters indicates that `T` may be constructed with `allocator_type` as its last constructor argument. Ideally, all constructors of

T (including the copy and move constructors) should have a variant that accepts a final argument of allocator type.

[Example:

```
template <class T, class A = allocator<T> >  
class Z {  
public:  
    typedef A allocator_type;  
  
    // Default constructor with optional allocator suffix  
    Z(const allocator_type& a = allocator_type());  
  
    // Copy constructor and allocator-extended copy constructor  
    Z(const Z& zz);  
    Z(const Z& zz, const allocator_type& a);  
};  
  
template <typename T>  
requires ConstructibleWithAllocatorSuffix<T, MyAlloc>  
struct MyContainer { };  
  
MyContainer<Z<int, MyAlloc> > C;
```

-- end example]

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

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

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

~~[Example:~~

```
template <class T, class A = allocator<T> >  
class Y {  
public:  
    typedef A allocator_type;  
  
    // Default constructor with and allocator-extended default constructor  
    Y();  
    Y(allocator_arg_t, const allocator_type& a);
```

```


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

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

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

end-example}


```

```

auto concept
ConstructibleWithAllocatorPrefix<class T, class Alloc,
                                class... Args>
: UsesAllocator<T, Alloc>
{
  requires HasConstructor<T, allocator arg t, Alloc, Args&&...>;
}

```

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

[Example:

```

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

  // Default constructor and allocator-extended default constructor
  Y();
  Y(allocator arg t, const allocator type& a);

  // Copy constructor and allocator-extended copy constructor
  Y(const Y& yy);
  Y(allocator arg t, const allocator type& a, const Y& yy);

  // Variadic constructor and allocator-extended variadic constructor
  template<class ...Args> Y(Args&&... args);
  template<class ...Args>
  Y(allocator arg t, const allocator type& a,

```



```

    Args&&... args);
};

template <typename T>
    requires ConstructibleWithAllocatorPrefix<T, MyAlloc,int>
    struct MyContainer { };

MyContainer<Z<int, MyAlloc> > C;

-- end example]

```

Add new concept and concept maps for ConstructibleWithAllocator:

The ConstructibleWithAllocator concept provides a uniform interface for passing an allocator to an object's constructor. There are concept maps to encompass almost all types, including those that don't use allocators at all. However, there is no concept map in this library for a type that uses the specified allocator, but which doesn't support passing the allocator to the specific constructor. [Note: The last restriction is to prevent the allocator being quietly ignored in a context where the user is likely to expect it to be used. – end note]

```

concept ConstructibleWithAllocator<class T, class Alloc,
                                class... Args> {
    T::T(allocator_prefix_t, Alloc, Args&&...);
}

template <Allocator Alloc, class T, class... Args>
    requires !UsesAllocator<T, Alloc>
        && HasConstructor<T, Args&&...>
    concept_map ConstructibleWithAllocator<Alloc, T, Args&&...> {

    T::T(allocator_arg_t, Alloc, Args&&...args)
        : T::T(forward<Args>(args)...) { }
}

template <Allocator Alloc, class T, class... Args>
    requires ConstructibleWithAllocatorPrefix<T, Alloc, Args&&...>
    concept_map ConstructibleWithAllocator<Alloc, T, Args&&...> {
}

template <Allocator Alloc, class T, class... Args>
    requires ConstructibleWithAllocatorSuffix<T, Alloc, Args&&...>
    concept_map ConstructibleWithAllocator<Alloc, T, Args&&...> {

    T::T(allocator_arg_t, const Alloc& a, Args&&...args)
        : T::T(forward<Args>(args)..., a) { }
}

```

The ConstructibleAsElement concept provides a uniform interface (construct_element – see section [construct.element]) for passing a scoped allocator's inner allocator to an object's constructor. There are concept maps to encompass almost all types, including those that don't use allocators at all. However, there is no concept map for a type that uses the

specified inner allocator, but which doesn't support passing the inner allocator to the specific constructor needed. [Note: ConstructibleAsElement differs from ConstructibleWithAllocator in that the former ignores the allocator unless it is scoped, and is constrained on the inner allocator, whereas the latter is constrained on the entire allocator and does not care whether or not it is scoped. – end note]

```

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

template <Allocator Alloc, class T, class... Args>
    requires !ScopedAllocator<Alloc>
        concept_map ConstructibleAsElement<Alloc,T,Args&&...> { }
template <ScopedAllocator Alloc, class T, class... Args>
    requires !UsesAllocator<T, Alloc::inner_allocator >
        concept_map ConstructibleAsElement<Alloc,T,Args&&...> { }
template <ScopedAllocator Alloc, class T, class... Args>
    requires ConstructibleWithAllocatorPrefix<T,
        Alloc::inner_allocator, Args&&...>
        concept_map ConstructibleAsElement<Alloc,T,Args&&...> { }
template <ScopedAllocator Alloc, class T, class... Args>
    requires ConstructibleWithAllocatorSuffix<T,
        Alloc::inner_allocator, Args&&...>
        concept_map ConstructibleAsElement<Alloc,T,Args&&...> { }

```

Allocator Propagation Concepts

Replace sections 20.6.3 [allocator.propagation] and 20.6.4 [allocator.propagation.map] with a single Allocator Propagation section:

~~20.6.4 allocator propagation map [allocator.propagation.map]~~

```

template <typename Alloc> struct allocator_propagation_map
{
    static Alloc select_for_copy_construction(const Alloc&);
    static void move_assign(Alloc& to, Alloc&& from);
    static void copy_assign(Alloc& to, const Alloc& from);
    static void swap(Alloc& a, Alloc& b);
};

```

20.6.4 Allocator propagation [allocator.propagation]

```

concept AllocatorPropagation<typename Alloc> {
    require Allocator<Alloc>;
    Alloc select_for_copy_construction(const Alloc&);
    void move_assign(Alloc& to, Alloc&& from);
    void copy_assign(Alloc& to, const Alloc& from);
    void swap(Alloc& a, Alloc& b);
}

```

~~Requires: Exactly one propagation trait shall derive from true_type for Alloc.~~

Remark: The `allocator_propagation_map` `AllocatorPropagation` concept provides functions to be used by containers for manipulating the allocators during construction, assignment, and swap operations. Each of the functions above have two possible behaviors. The implementations the specific behavior of functions above for this concept are dependent on the allocator propagation traits of the which of four refinements of this concept is mapped for a specific Alloc.

```
Alloc select_for_copy_construction(const Alloc& x);
```

returns: ~~if `allocator_propagate_never`<Alloc>~~ if allocator should be propagated on copy and move construction, returns Alloc(), else returns x;

```
void move_assign(Alloc& to, Alloc&& from);
```

effects: ~~if `allocator_propagate_on_move_assignment`<Alloc> or `allocator_propagate_on_copy_assignment`<Alloc>~~ if allocator should be propagated on move assignment, assign to = forward(from), otherwise do nothing.

```
void copy_assign(Alloc& to, const Alloc& from);
```

effects: ~~if `allocator_propagate_on_copy_assignment`<Alloc>~~ if allocator should be propagated on copy assignment, assign to = from, otherwise do nothing.

```
void swap(Alloc& a, Alloc& b);
```

effects: ~~if `allocator_propagate_on_move_assignment`<Alloc> or `allocator_propagate_on_copy_assignment`<Alloc>~~ if allocator should be propagated on move assignment, exchange the values of *a* and *b*. Otherwise, if *a* == *b*, do nothing. Otherwise the behavior is undefined. [Footnote – Implementations are encouraged to diagnose the undefined behavior at runtime using assert or a similar mechanism for debug/non-optimized executions – end footnote]

```
template <typename Alloc>
struct allocator_propagate_never : false_type { };
concept AllocatorPropagateNever<typename Alloc>
: AllocatorPropagation<Alloc> {
    Alloc select_for_copy_construction(const Alloc& x)
    { return Alloc(); }
    void move_assign(Alloc& to, Alloc&& from) { }
    void copy_assign(Alloc& to, const Alloc& from) { }
    void swap(Alloc& a, Alloc& b) { }
}
```

requires: Alloc is an Allocator

remark: if ~~specialized to derive from true_type~~ a concept map is defined for `AllocatorPropagateNever`<Alloc> for a specific allocator type, it indicates that a container using the specified Alloc should not copy or move the allocator when the container is copy-constructed, move-constructed, copy-assigned, moved-assigned, or swapped. It is unspecified whether the allocator is propagated in situations where this standard allows copy construction to be elided.

```
template <typename Alloc>
struct allocator_propagate_on_copy_construction : see below
concept AllocatorPropagateOnCopyConstruction<typename Alloc>
: AllocatorPropagation<Alloc> {
    Alloc select_for_copy_construction(const Alloc& x)
    { return x; }
}
```

```

void move assign(Alloc& to, Alloc&& from) { }
void copy assign(Alloc& to, const Alloc& from) { }
void swap(Alloc& a, Alloc& b) { }
}

```

requires: Alloc is an Allocator

remark: if ~~specialized to derive from true_type~~ a concept map is defined for `AllocatorPropagateOnCopyConstruction<Alloc>` for specific allocator type, it indicates that a container using the specified Alloc should copy or move the allocator when the container is copy-constructed or move-constructed, but not when the container is copy-assigned, moved-assigned, or swapped.

default: ~~The unspecialized trait derive from true_type if none of allocator_propagate_never, allocator_propagate_on_move_assignment, or allocator_propagate_on_copy_assignment are derived from true_type for the given Alloc type. Otherwise, it derives from false_type. A concept map AllocatorPropagateOnCopyConstruction<Alloc> is defined automatically if Alloc does not model AllocatorPropagateNever<Alloc>, AllocatorPropagateOnMoveAssignment<Alloc>, or AllocatorPropagateOnCopyAssignment<Alloc>.~~

```

template <typename Alloc>
struct allocator_propagate_on_move_assignment : false_type { };
concept AllocatorPropagateOnMoveAssignment<typename Alloc>
: AllocatorPropagation<Alloc> {
    Alloc select for copy construction(const Alloc& x)
    { return x; }
    void move assign(Alloc& to, Alloc&& from)
    { to = forward(from); }
    void copy assign(Alloc& to, const Alloc& from) { }
    void swap(Alloc& a, Alloc& b)
    { swap a and b }
}

```

requires: Alloc is an Allocator

remark: if ~~specialized to derive from true_type~~ a concept map is defined for `AllocatorPropagateOnMoveAssignment<Alloc>` for specific allocator type, it indicates that a container using the specified Alloc should copy or move the allocator when the container is copy-constructed, move-constructed, moved-assigned, or swapped but not when the container is copy-assigned.

```

template <typename Alloc>
struct allocator_propagate_on_copy_assignment : false_type { };
concept AllocatorPropagateOnCopyAssignment<typename Alloc>
: AllocatorPropagation<Alloc> {
    Alloc select for copy construction(const Alloc& x)
    { return x; }
    void move assign(Alloc& to, Alloc&& from)
    { to = forward(from); }
    void copy assign(Alloc& to, const Alloc& from)
    { to = from; }
}

```

```

    void swap(Alloc& a, Alloc& b)
    { swap a and b }
}

```

requires: Alloc is an Allocator

remark: if specialized to derive from true_type a concept map is defined for AllocatorPropagateOnCopyAssignment<Alloc> for specific allocator type, it indicates that a container using the specified Alloc should copy or move the allocator when the container is copy-constructed, move-constructed, moved-assigned, swapped or copy-assigned.

Scoped Allocator Adaptor

In section 20.6.6 [allocator.adaptor], constraint the `scoped_allocator_adaptor` template so that its arguments model the Allocator concept. Also, add definitions and use of `generic_pointer`:

```

namespace std {

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

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

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

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

            scoped_allocator_adaptor();
            scoped_allocator_adaptor(scoped_allocator_adaptor&&);
            scoped_allocator_adaptor(const scoped_allocator_adaptor&);

```

```

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

template <typename Allocator OuterA2>
    requires Convertible<OuterA2&&, OuterA>
    scoped_allocator_adaptor(
        scoped_allocator_adaptor<OuterA2, void>&&);
template <typename Allocator OuterA2>
    requires Convertible<const OuterA2&, OuterA>
    scoped_allocator_adaptor(
        const scoped_allocator_adaptor<OuterA2, void>&);

~scoped_allocator_adaptor();

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

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

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

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

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

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

        template <typename ObjectType _Tp>
        struct rebind
        {
            typedef scoped_allocator_adaptor<OuterA::template rebind<_Tp>::other,

```

```

        Allocator<OuterA>::rebind< Tp>,
        InnerA> other;
};

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

template <typename Allocator OuterAlloc2>
    requires Convertible<OuterA2&&, OuterA>
    scoped_allocator_adaptor(
        scoped_allocator_adaptor<OuterAlloc2&, InnerA>&&);
template <typename Allocator OuterAlloc2>
    requires Convertible<const OuterA2&, OuterA>
    scoped_allocator_adaptor(
        const scoped_allocator_adaptor<OuterAlloc2&, InnerA>&);

~scoped_allocator_adaptor();

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

pointer allocate(size_type n);
template <typename HintP>
    pointer allocate(size_type n, HintP const generic pointer u);

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

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

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

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

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

```

```
}
```

Repeat the above changes for the individual function descriptions:

```
template <typename Allocator OuterA2>
    requires Convertible<OuterA2&&, OuterA>
    scoped_allocator_adaptor(
        scoped_allocator_adaptor<OuterA2, InnerA>&& other);
template <typename Allocator OuterA2>
    requires Convertible<OuterA2&&, OuterA>
    scoped_allocator_adaptor(
        const scoped_allocator_adaptor<OuterA2, InnerA>& other);

template <class... Args>
    requires HasConstructor<value type, Args&&...>
    void construct(pointer p, Args&&... args);

    effects: outer_allocator().construct(p, forward<Args>(args)...);

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

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

Element construction

Replace most of 20.6.9 [construct.element], as follows:

20.6.9 construct_element [construct.element]

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

[*Note:* **This** The appropriate overload of the construct_element function is called from within containers in order to construct elements during insertion operations as well as to move elements during reallocation operations. It automates the process of determining if the scoped allocator model is in use and transmitting the inner allocator for scoped allocators. – *end note*]

Effects: ~~The first of the following items that applies:~~

- ~~— if `scoped_allocator<Alloc>` is derived from `false_type` or uses `allocator<T, A::inner_allocator_type>` is derived from `false_type`, `alloc.construct(alloc.address(r), args...)`~~
- ~~— if `constructible_with_allocator_prefix<T, A::inner_allocator_type, Args...>` is derived from `true_type`, `alloc.construct(alloc.address(r), allocator_arg_t, alloc.inner_allocator(), args...)`~~

- ~~if `constructible_with_allocator_suffix`<T, A::inner_allocator_type, Args...> is derived from `true_type`, `alloc.construct(alloc.address(r), args..., alloc.inner_allocator())`~~
- ~~otherwise, the program is ill-formed. [Note: The `ConstructibleAsElement` constraint ensures that this cannot occur at runtime~~

And add the following:

```
template <Allocator Alloc, class T, class... Args>
requires !ScopedAllocator<Alloc>
        && HasConstructor<T, Args&&...>
        void construct_element(Alloc& a, T& r, Args&&... args);

Effects: a.construct(a.address(r), forward<Args>(args)...)

template <ScopedAllocator Alloc, class T, class... Args>
requires !UsesAllocator<T, Alloc::inner_allocator>
        && HasConstructor<T, Args&&...>
        void construct_element(Alloc& a, T& r, Args&&... args);

Effects: a.construct(a.address(r), forward<Args>(args)...)

template <ScopedAllocator Alloc, class T, class... Args>
requires ConstructibleWithAllocatorPrefix<T,
        Alloc::inner_allocator, Args&&...>
        void construct_element(Alloc& a, T& r, Args&&... args);

Effects: a.construct(a.address(r), allocator_arg_t,
        alloc.inner_allocator(), forward<Args>(args)...)

template <ScopedAllocator Alloc, class T, class... Args>
requires ConstructibleWithAllocatorSuffix<T,
        Alloc::inner_allocator, Args&&...>
        void construct_element(Alloc& a, T& r, Args&&... args);

Effects: a.construct(a.address(r), forward<Args>(args)...,
        alloc.inner_allocator())
```

Acknowledgements

Thank you to Doug Gregor for his invaluable assistance with concepts. Thank you to John Lakos for his support, guidance, and encouragement. Thank you to Lance Diduck for suggesting, among other things, that not all allocators need to provide random access to allocated memory.

References

All documents referenced here can be found at
<http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2008/>.

[N2641](#): Allocator Concepts

[N2554](#): The scoped allocator model (Rev 2)

[N2525](#): Allocator-specific move and swap

[N2621](#): Core Concepts for the C++0x Standard Library

[N2623](#): Concepts for the C++0x Standard Library: Containers