Doc No: N2768=08-0278
Date: 2008-09-19
Author: Pablo Halpern
Bloomberg, L.P.

phalpern@halpernwightsoftware.com

Allocator Concepts, part 1 (revision 2)

Contents

Summary ... 1
Changes from N2654 ... 2
  List of changes ... 2
  Future Proposals ... 2
Document Conventions ... 3
Proposed Wording ... 3
  The addressof Function ... 3
  Header changes ... 4
  Allocator Concept ... 5
  Allocator-related Element Concepts ... 10
  Scoped Allocator Adaptor ... 14
  Element construction ... 17
Acknowledgements ... 18
References ... 18

Summary

This paper defines concepts for Allocator and a number of related requirements and specifies concept constraints and concept maps for a number of library classes and class templates. Most of these concepts are used in N2738: *Concepts for the C++0x Standard Library: Containers* and N2735: *Concepts for the C++0x Standard Library: Utilities*. 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 are 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. A default implementation of `Allocator<X>::construct()` allows C++03 allocators to work as C++0x allocators by automatically providing a variadic `construct()` function.

Changes from N2654

This document is mostly a subset of N2654, Allocator Concepts (rev 1). It comprises those parts of the Allocator Concepts paper that we feel might reasonably move forward at the September 2008 meeting in San Francisco and provides "place holder" concepts for completing the work of conceptifying the allocator requirements. In particular, any allocator-related concept that is required by the utilities or containers sections is represented either in whole or as a placeholder. Specific details needed to implement the scoped allocator model (N2554) and allocator-specific move and swap (N2525) have been left out of this document and deferred to a future (part 2) document.

List of changes

- Eliminated RandomAccessAllocator, SimpleAllocator, and MinimalAllocator for now.
- Removed machinery for creating concept maps for ConstructibleWithAllocator.
- Renamed ConstructibleAsElement to AllocatableElement
- Eliminated the ScopedAllocator concept. Moved scoped allocator dispatch into the AllocatableElement concept.
- Eliminated the allocator propagation concepts – We will find a cleaner way of making that mechanism work.
- Changed construct(pointer, args…) and destroy(pointer) to construct(value_type*,args…) and destroy(value_type*) in the Allocator concept.
- Added HasAllocatorType concept
- Added requirement that pointer and const_pointer be Regular and that none of the regular operations throw an exception.

Future Proposals

In addition to a proposal conceptifying scoped allocators and allocator propagation, I expect to submit proposals in time for the next meeting addressing the following features that were originally proposed in N2654:

1. Not all allocators require allocation of elements at once. The Allocator concept can be split into two concepts, a basic one that allocates single objects and does not require random-access pointer types, and a refinement that allocates memory for multiple, contiguous elements and for which the pointer type is a random-access iterator. (An in-between concept that allocates multiple objects and provides only forward-iterator pointers is possible if reasonable use cases can be demonstrated.)

2. User-defined container types that don't want to deal with non-raw pointer types, etc., could be constrained with a `SimpleAllocator` concept, which would require that the allocator use raw pointers.

3. An allocator author could allow a container to bypass calls to `construct()` and `destroy()` by somehow specifying that the allocator doesn't do anything special in those functions. 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 (`std::allocator`) would fall into this category.

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.

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

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

  concept ConstructibleWithAllocator<class T, class Alloc,
                                     class... Args> see below
  template <Allocator Alloc, class T, class... Args>
    requires Unspecified
      concept_map ConstructibleWithAllocator<T, Alloc, Args&&...> see below

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

  template <Allocator Alloc, class T, class... Args>
    requires HasConstructor<T, Args&&...>
      concept_map AllocatableElement<Alloc,T,Args&&...> see below
}
```

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;
```

The `is_scoped_allocator` trait remains until the next proposal, when it will be replaced by a different, concept-based mechanism for identifying and handling scoped allocators.

Also concept maps and allocator-related constraints:

```
// 20.6.5, the default allocator:
template <class T> class allocator;
template <ObjectType T>
```

```
  concept_map Allocator<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 { };
template < classAllocator OuterA, classAllocator InnerA>
  struct allocator_propagate_never<scoped_allocator_adaptor<OuterA,
InnerA> >
    : true_type { };
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 <classAllocator Alloc, class T, class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
    void construct_element(Alloc& alloc, T& r, Args&&... args);
```

Allocator Concept

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. ~~All the operations on the allocators are expected to be amortized constant time.~~ ~~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;
    requires Regular<pointer>
          && RandomAccessIterator<pointer>
          && Regular<const_pointer>
          && RandomAccessIterator<const_pointer>;
    SignedIntegralLike difference_type =
        RandomAccessIterator<pointer>::difference_type;
    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(value_type* p, Args&&... args)
    {
        ::new ((void*) p) value_type(forward<Args>(args)...);
    }

    void X::destroy(value_type* 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. The behavior is undefined if an exception is propagated when applying any operation from the `Regular` concept to a `pointer, const_pointer, generic_pointer,` or `const_generic_pointer`.

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.

```
SignedIntegralLike difference_type;
```

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

```
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 `SameAllocator<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 – *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, a pointer to 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(value_type* p, Args&&... args);
```

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

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

```
void X::destroy(value_type* p);
```

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

Default: `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*`.

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 `HasAllocatorType` and `UsesAllocator` concepts:

```
auto concept HasAllocatorType<typename T>
{
    typename allocator_type = T::allocator_type;
    requires Allocator<allocator_type>;
}
```

Remark: Automatically detects if `T` has a nested `allocator_type` that meets the requirements of an allocator.

~~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 Allocator<allocator_type> &&
             Convertible<Alloc, allocator_type>;
}
```

Remark: Automatically detects if `T` has a nested `allocator_type` that is convertible from `Alloc`. ~~Meets the BinaryTypeTrait requirements ([meta.rqmts] 20.4.1).~~ A program may ~~specialize this type to derive from true_type~~ 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`. [*Note:* Although the default concept maps for these concepts concepts often causes them to appear in pairs, `UsesAllocator` does not imply `HasAllocatorType`, nor vice versa. Similarly, the !`UsesAllocator` does not imply !`HasAllocatorType`, nor vice versa.]

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

Remove `constructible_with_allocator_suffix/prefix` traits:

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

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

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

~~[*Example:*~~

```
  template <class T, class A = allocator<T> >
  class Z {
```

```
    public:
        typedef A allocator_type;

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

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

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

-- *end example*]

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

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

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

[*Example:*

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

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

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

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

```
            Args&&... args);
    };
```

~~// Specialize trait for class template Y~~

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

~~– end example~~]

Add new concepts and concept maps for `ConstructibleWithAllocator` and `AllocatableElement`:

`The ConstructibleWithAllocator` concept provides a uniform interface for passing an allocator to an object's constructor.

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

The library defines concept map templates to adapt `ConstructibleWithAllocator` for each pattern of constraints described in table *xyz*. Each concept map adapts `T`'s constructor, mapping the variadic argument pack from its position in the `ConstructibleWithAllocator` concept into its corresponding position in the actual constructor for `T`, and mapping the `Alloc` and `allocator_arg_t` arguments to their appropriate positions (if any) in the argument list for `T`'s constructor. The concept maps shall be constrained such that, in situations where a set of types matches more than one pattern, the partial ordering of concept maps gives precedence to those patterns described earlier in the table. [*Note:* 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 *an* allocator, but that doesn't support passing the *specified* allocator to the *specified* constructor. 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*]

Table *xyz*: `ConstructibleWithAllocator` concept map constraint patterns

Concept requirements	Constructor requirement
`UsesAllocator<T, Alloc>`	`T::T(allocator_arg_t, Alloc, Args&&...);`
`UsesAllocator<T, Alloc>`	`T::T(Args&&..., Alloc);`
`!HasAllocatorType<T> && !UsesAllocator<T,Alloc>`	`T::T(Args&&...);`

`The AllocatableElement` concept provides a uniform interface (`construct_element` – see section [construct.element]) for constructing an object obtained from an allocator. A concept map provides a default implementation that is suitable for most allocators. Specific allocator templates may provide more specialized concept maps (see [allocator.adaptor]) for an example). [*Note*: `ConstructibleWithAllocator` differs from `AllocatableElement` in that the former describes how to construct an item that *uses* an allocator whereas the latter describes how to construct an item that *was allocated from* an allocator – *end note*]

```
concept AllocatableElement<class Alloc, class T, class... Args>
```

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

template <Allocator Alloc, class T, class... Args>
  requires HasConstructor<T, Args...>
concept_map AllocatableElement<Alloc, T, Args&&...>
{
    void construct_element(Alloc& a, T* t, Args&&... args)
      { Alloc::rebind<T>(a).construct(t, forward(args)...); }
}
```

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< typenameAllocator OuterA, typenameAllocator InnerA =
voidunspecified allocator type>
    class scoped_allocator_adaptor ;

  template< typenameAllocator OuterA>
    class scoped_allocator_adaptor<OuterA, void unspecified allocator type> :
      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 <typenameObjectType _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 <typenameAllocator OuterA2>
   requires Convertible<OuterA2&&, OuterA>
    scoped_allocator_adaptor(
        scoped_allocator_adaptor<OuterA2, void>&&);
  template < typenameAllocator 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, _HintPconst_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 <typenameObjectType _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 <typenameAllocator OuterAlloc2>
     requires Convertible<OuterA2&&, OuterA>
      scoped_allocator_adaptor(
        scoped_allocator_adaptor<OuterAlloc2&,InnerA>&&);
    template <typenameAllocator 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, _HintPconst 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(pointervalue type* p, Args&&... args);
    void destroy(pointervalue type* p);

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

  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);

  template <Allocator OuterA, Allocator InnerA,
            typename T, typename... Args>
    concept_map AllocatableElement<
        scoped_allocator_adaptor<OuterA,InnerA>, T, Args&&...>
      ConstructibleWithAllocator<T, InnerA, Args&&...>
    {
        void construct_element(
            scoped_allocator_adaptor<OuterA,InnerA>& alloc,
            T* p, Args&&... args)
        {
            OuterA::rebind<T> outer = alloc.outer_allocator();
            InnerA& inner = alloc.inner_allocator();
            outer.construct(allocator_arg_t(), inner, forward(args)...);
        }
    }
}
```

Repeat the above changes for the individual function descriptions:

```
template <Allocator OuterA2>
 requires Convertible<OuterA2&&, OuterA>
  scoped_allocator_adaptor(
       scoped_allocator_adaptor<OuterA2, InnerA>&& other);
template <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(value_type* p, Args&&... args);
```

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

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

template<Allocator OuterA1, Allocator OuterA2, 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 is_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 AllocatableElement constraint ensures that this cannot occur at runtime~~

And add the following:

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

Effects: `AllocatableElement<Alloc, T, Args&&...>::construct_element(a, addressof(r), forward<Args>(args)...)`

The global construct_element will almost certainly disappear as we finish conceptifying the scoped allocator model.

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. Last but not least, thanks to David Abrahams for his help in simplifying this proposal during the course of the San Francisco meeting and helping me lay the foundation for part 2 (for the next meeting).

References

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

N2654: Allocator Concepts (Rev 1)

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