Doc No: N2810=08-0320
Date: 2008-12-01
Author: Pablo Halpern
Cilk Arts, Inc.

phalpern@halpernwightsoftware.com

Defects and Proposed Resolutions for Allocator Concepts

Contents

Summary

This paper is an exposition of a comment submitted in response to the C++0x CD. The addition of concepts for allocators in the standard library is incomplete and has a number of defects. Each defect is listed below with a proposed resolution. For easier reference, most of the proposed changes are aggregated into a modified `Allocator` concept (Appendix A) and a modified `scoped_allocator_adaptor` (Appendix B) at the end. Proposed resolutions do not include detailed clause-by-clause changes, but are intended to be sufficiently detailed to produce such a clause-by-clause description (i.e., in a future paper).

Document Conventions

All section names and numbers are relative to the October 2008 working draft, N2798.

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).

`Allocator` concept does not match all C++03 allocators

Description of Issue

The reason that `Allocator` is declared `auto` is to support backwards compatibility with C++03 allocators. Otherwise, the `Allocator` concept is not the kind of concept that would normally be declared `auto`. However, declaring it `auto` does not fully succeed at providing this backwards compatibility. C++03 allocators do not have a `generic_pointer` type nor a variadic `construct` function. Moreover, the non-variadic `construct` function takes a `pointer` argument instead of a `T*` argument as in the concept. As a result, there are some C++03 allocators for which there would be no automatically-generated concept map.

Proposed Resolution

Remove the `auto` modifier from the `Allocator` concept. This change will make `Allocator` cleaner (by avoiding gratuitous use of `auto`) and will allow additional evolution without the constraint of automatic compatibility with C++03 allocators. To address the compatibility problem, require that there be a `concept_map` for `Allocator` that automatically adapts any class meeting the requirements of a C++03 allocator.

Allocator for `vector` and `string` are Under-constrained

Description of Issue

The `vector` and `string` templates both expose the fact that their elements are contiguous in memory. Both templates have `data()` methods that return a value equivalent to the address of their first element. It can be deduced, therefore, that `Alloc::pointer` must be a raw pointer to the value type and that allocators using unusual pointer types will not work with `vector` or `string`.

Proposed Resolution

Add concept constraints to `vector` and `string` requiring that their allocators use raw pointers. I.e.:

```
template <ValueType T, Allocator Alloc = allocator<T> >
requires SameType<Alloc::pointer, T*>
      && SameType<Alloc::const_pointer, const T*>
class vector { ... };
```

`Allocator::rebind` is different from C++03 `rebind`

Description of Issue

The `rebind` template in the `Allocator` concept serves the same purpose as the `rebind` template in the C++03 allocator requirements, but uses a different syntax for both definition and declaration. In the case of a C++03 allocator, rebinding is done by referencing `Alloc::rebind<U>::other`, whereas in C++0x, rebinding is done by referencing: `Allocator<Alloc>::rebind<U>` (no `other` nested type). Moreover, the declaration of `rebind` in a C++0x concept map is simpler than in a C++03 concept: `template <class T> rebind = MyAlloc<T>;` instead of `template <class T> struct rebind { typedef MyAlloc<T> other; };`. These differences are confusing and could cause strange compilation errors when adding constraints to an unconstrained container template.

Proposed Resolution

Rename the `rebind` template in the `Allocator` concept to something else. Some ideas are `retype`, `related_instance`, `sibling`, or `sibling_allocator`. Optionally, add a `rebind` template that works like C++03 `rebind` and is implemented in terms of `sibling_allocator`. The default `related_instance` template can still be implemented in terms of `rebind`.

Construct Method is Limited to `value_type`

Description of Issue

The `construct` method in the `Allocator` concept only constructs objects of type `value_type`. This constraint can lead to inconvenient and sometimes inefficient uses of `rebind` in order to construct objects of different types. For example, a container type might allocate objects of type `Node<T>` from an allocator, `alloc_`, of type `allocator_type == Alloc<Node<T>>`. However, some parts of the `Node` might be initialized independently of the `T` object contained within it. Initializing the inner object would require the use of `rebind` as follows:

```
allocator_type::rebind<T>(alloc_).construct(p, args);
```

The above construct is not only hard to read, but it constructs a temporary object of type `Alloc<T>` just to call its `construct` method.

Proposed Resolution

(Note: See an alternative resolution in the next issue.) We have already changed `construct` to a template in order to support `emplace`. It is a small matter, then, to templatized the pointer argument as well as the constructor arguments:

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

Note that the rare allocator requiring a very different implementation of `construct` for each different data type can use `rebind` internally in its implementation of `construct`.

`construct_element` Function is Unnecessary

Description of Issue

The `construct_element` function was originally introduced to dispatch the construction of elements based on whether the allocator was a scoped allocator. The advent of concepts allows us to add requirements to the `construct` member function of each allocator, removing the need for a separate `construct_element` dispatch function.

Proposed Resolution

Remove the global `construct_element` template and remove the `construct_element` function from the `AllocatableElement` concept. Move the `construct` function from the `Allocator` concept to the `AllocatableElement` concept, as shown in Appendix A. This change will also lift the restriction on `construct` described in the previous issue. Change other uses of `construct_element` to use `construct` directly.

This change, however, will remove the default implementation of `construct` because such a default implementation would potentially bypass constraints required by an advanced allocator (such as a scoped allocator). There are at least two ways regain the default implementation of `construct`: 1) Provide a base class containing the most common elements of an allocator, including a default implementation of `construct`, as a starting point for most allocator implementations. 2) Add a new `AdvancedAllocator` (name open to change) refinement to `Allocator` that is used in the negative by a concept map to produce a default implementation. This solution (shown in Appendix A) has the additional advantage that the `AdvancedAllocator` concept can be the basis of some optimizations (e.g., memcpy can be used for POD types allocated from a non-advanced allocator).

See the Appendix A for an embodiment of this proposed resolution.

`is_scoped_allocator` Trait is not Used

Description of Issue

The `is_scoped_allocator` trait is still in the WP, but is no longer used since allocator concepts were introduced. The function of the `is_scoped_allocator` trait, to dispatch the `construct_element` functionality, is not handled anywhere.

Proposed Resolution

Remove the `is_scoped_allocator` trait. Constrain the `construct` member function of any scoped allocator such that an element must be `ConstructibleWithAllocator` using the inner allocator type.

See the Appendix B for an embodiment of this proposed resolution.

`scoped_allocator_adaptor` has errors

Description of Issue

The `scoped_allocator_adapator` templates have errors in them, some of which cause them not to model the `Allocator` concept in every detail. Specifically:

- There are places where `void` is used instead of *unspecified allocator type*.
- The `construct` and `destroy` methods take `pointer` instead of `value_type*`. (But if the resolution to the `construct` issue is accepted, it should take a pointer to template-argument type.)
- Some `Allocator` constraints are missing.

Proposed Resolution

Correct the errors. See related issues in this paper for other changes that may apply.

See the Appendix B for an embodiment of this proposed resolution.

Two Types of `scoped_allocator_adaptor`s

Description of Issue

The WP describes `scoped_allocator_adaptor` as a class template with two template parameters, one for the outer allocator type and one for the inner allocator type. A specialization of `scoped_allocator_adaptor` takes only one template parameter. In the latter case, not only are both the outer and inner allocators the same type, they are also the same object. Thus there is a difference between `scoped_allocator_adaptor<A1, A1>`, which holds two distinct instances of type `A1`, and `scoped_allocator_adaptor<A1>`, which holds a single instance of type `A1`. This use of a default parameter has already caused significant confusion.

Proposed Resolution

Use separate names, `scoped_allocator_adaptor` and `scoped_allocator_adaptor2` for the single-parameter and dual-parameter adaptor templates, respectively. Remove the default argument of *unspecified allocator type*.

See the Appendix B for an embodiment of this proposed resolution.

No Concepts for Allocator Propagation

Description of Issue

The allocator propagation traits, `allocator_propagate_never`, `allocator_propagate_on_copy_construction`, `allocator_propagate_on_move_assignment`, and `allocator_propagate_on_copy_assignment` are pre-concept ways to express what can now be expressed with concepts. These traits control the behavior of allocator propagation for a set of static member functions in the `allocator_propagation` structure. The system is unnecessarily complex now that we have an `Allocator` concept into which we can directly insert the four propagation functions with default implementations.

Proposed Resolution

Add the following four functions and default implementations to the `Allocator` concept:

```
Alloc select_on_container_copy_construction(const Alloc& x) { return x; }
Alloc select_on_container_move_construction(Alloc&& x) { return x; }
void do_on_container_copy_assignment(Alloc& to, Alloc& from) { }
void do_on_container_move_assign(Alloc& to, Alloc&& from) { }
void do_on_container_swap(Alloc& a, Alloc& b) { }
```

Individual allocator types can override these defaults as desired.

See the Appendix A for an embodiment of this proposed resolution.

Too Many `pair` Constructors

Description of Issue

The addition of rvalue constructors, variadic constructors, and allocator constructors to `pair` has increased the number of explicitly-specified constructors from 3 to 9, making an otherwise simple class rather arcane.

Proposed Resolution

It should be possible to create a concept for constructing any type from a `tuple` or `tuple`-like object containing the type's constructor arguments. A `pair` constructor could be added that accepts two such "packaged" constructor arguments and passes each one to the constructors of `first` and `second` accordingly. We could then remove the allocator constructors from `pair` and instead move them into specializations of `ConstructibleWithAllocator`.

Separately, we could consider removing the variadic constructors from `pair` by allowing containers like `map` and `unordered_map` to construct `first` and `second` independently.

Appendix A: Modified Allocator Concepts

Although this section does not describe clause-for-clause changes to the draft, the concept changes below provide a more formal description of most of the proposed resolutions described above.

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

    ObjectType value_type = typename X::value_type;
    ~~DereferenceableHasDereference~~ pointer = see below;
    ~~DereferenceableHasDereference~~ 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~~related instance = 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~~related_instance<value_type>, X>;
    requires SameType<generic_pointer
                      ,~~rebind~~related_instance<unspecified unique
type>::generic_pointer>;
        // see description of generic_pointer, below
    requires SameType<const_generic_pointer
                  ,~~rebind~~related_instance< 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~~related_instance<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
    }

    Alloc select_on_container_copy_construction(const Alloc& x)
        { return x; }
    Alloc select_on_container_move_construction(Alloc&& x) { return x; }
    void do_on_container_copy_assignment(Alloc& to, Alloc& from) { }
    void do_on_container_move_assign(Alloc& to, Alloc&& from) { }
```

```
    void do_on_container_swap(Alloc& a, Alloc& b) { }
}

template <typename Alloc>
requires Some auto concept that matches C++03 Allocators
concept map Allocator<Alloc> {
    Map C++0x allocator functionality onto C++03 allocator
}

concept AdvancedAllocator<typename X> : Allocator<X>
{
}

auto concept AllocatableElement<class Alloc, class T, class... Args>
{
    requires Allocator<Alloc>;
    void ~~construct_element(Alloc& a,~~
      Alloc::construct(T* t, Args&&... args);
}

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

Appendix B: Modified `scoped_allocator_adaptor`

```
  template<Allocator ~~OuterA~~Alloc>
    class scoped_allocator_adaptor<~~ OuterA~~Alloc~~, unspecified allocator type~~> :
      public ~~OuterA~~Alloc
  {
  public:
    typedef ~~OuterA~~Alloc outer_allocator_type;
    typedef ~~OuterA~~Alloc inner_allocator_type;
        // outer and inner allocator types are the same.

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

    template <ObjectType _Tp>
```

```
    struct rebind
    {
        typedef scoped_allocator_adaptor<
           Allocator<~~OuterA~~Alloc >::~~rebind~~related_instance<_Tp>~~, void~~
        > other;
    };

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

    template <Allocator ~~OuterA~~Alloc2>
     requires Convertible<~~OuterA~~Alloc2&&, ~~OuterA~~Alloc>
      scoped_allocator_adaptor(
          scoped_allocator_adaptor<~~OuterA~~Alloc2~~, void~~>&&);
    template <Allocator ~~OuterA~~Alloc2>
     requires Convertible<const ~~OuterA~~Alloc2&, ~~OuterA~~Alloc>
      scoped_allocator_adaptor(
          const scoped_allocator_adaptor<~~OuterA~~Alloc2~~, void~~>&);

   ~scoped_allocator_adaptor();

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

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

    template <class... Args>
     requires ~~HasConstructor~~
       ConstructibleWithAllocator<value_type, Alloc, Args&&...>
      void construct(pointer p, Args&&... args);
    template <Allocator Alloc2, class... Args>
     requires ConstructibleWithAllocator<value_type, Alloc2, Args&&...>
      void construct(value_type* p, allocator_arg_t,
                     const Alloc2&, Args&&... args);
    void destroy(pointer p);

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

  template<~~typename~~Allocator OuterA, ~~typename~~Allocator InnerA>
    class scoped_allocator_adaptor2 : public OuterA
  {
  public:
    typedef OuterA outer_allocator_type;
    typedef InnerA inner_allocator_type;
```

```
typedef typename Allocator<outer_allocator_type>::size_type       size_type;
typedef typename Allocator<outer_allocator_type>::difference_type difference_type;
typedef typename Allocator<outer_allocator_type>::pointer         pointer;
typedef typename Allocator<outer_allocator_type>::const_pointer   const_pointer;
typedef typename Allocator<outer_allocator_type>::generic_pointer generic_pointer;
typedef typename Allocator<outer_allocator_type>::const_generic_pointer
                                                        const_generic_pointer;
typedef typename Allocator<outer_allocator_type>::reference       reference;
typedef typename Allocator<outer_allocator_type>::const_reference const_reference;
typedef typename Allocator<outer_allocator_type>::value_type      value_type;

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

 scoped_allocator_adaptor2();
 scoped_allocator_adaptor2(outer_allocator_type&& outerAlloc,
                           inner_allocator_type&& innerAlloc);
 scoped_allocator_adaptor2(const outer_allocator_type& outerAlloc,
                           const inner_allocator_type& innerAlloc);
 scoped_allocator_adaptor2(scoped_allocator_adaptor&& other);
 scoped_allocator_adaptor2(const scoped_allocator_adaptor& other);

 template <Allocator OuterA2>
  requires Convertible<OuterA2&&, OuterA>
   scoped_allocator_adaptor(
     scoped_allocator_adaptor2<OuterA2&,InnerA>&&);
 template <Allocator OuterA2>
  requires Convertible<const OuterA2&, OuterA>
   scoped_allocator_adaptor(
     const scoped_allocator_adaptor2<OuterA2&,InnerA>&);

~scoped_allocator_adaptor2();

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

 pointer allocate(size_type n);
   pointer allocate(size_type n, const_generic_pointer u);

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

 template <class... Args>
  requires ~~HasConstructor~~
    ConstructibleWithAllocator<value_type, Alloc, Args&&...>
   void construct(value_type* p, Args&&... args);
 template <Allocator Alloc2, class... Args>
```

```
    requires ConstructibleWithAllocator<value_type, Alloc2, Args&&...>
     void construct(value_type* p, allocator_arg_t,
                    const Alloc2&, Args&&... args);
   void destroy(value_type* p);

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

 template <Allocator Alloc>
 concept_map AdvancedAllocator<scoped_allocator_adaptor<Alloc> > {
 }

 template <Allocator A1, Allocator A2>
 concept_map AdvancedAllocator<scoped_allocator_adaptor2<A1, A2> > {
 }
```

References

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

N2768: Allocator Concepts, part 1 (revision 2)

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