Doc No: N2943=09-0133
Date: 2009-07-30
Author: Pablo Halpern
Cilk Arts, Inc.

phalpern@halpernwightsoftware.com

Allocators without Concepts (preview)

Contents

Motivation and Summary

The adoption of N2554 (The Scoped Allocator Model) and N2525 (Allocator-specific Swap and Move Behavior) in Belevue (February/March 2008) made allocators much more useful and flexible than they were in 1998. It has been pointed out, however, that these improvements came at the cost of some interface complexity. Of particular concern was the fact that the presence of scoped allocators required the definition and testing of traits in numerous places in the standard library.

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

This paper represents the first step towards bringing as many of the concepts-based simplifications to the allocator library as possible in the absence of concepts. This is not a formal proposal and it does not contain formal wording. My intention, rather, is to provide a "heads up" to the library working group outlining a possible direction for allocators and providing a basis for feedback in advance of a formal proposal. Expect a formal proposal before the Santa Cruz meeting (October, 2009). This paper does not address the issue of constructors to pair. That issue will be addressed by a separate revision of N2908.

The purpose of this paper is two-fold: 1) to assure the committee that the simplifications in the allocator system that were described in N2768 and N2840 will not be lost, and 2) to solicit early feedback so that a solid proposal can be brought to Santa Cruz and passed in the same meeting.

National Body Comments Addressed in this Paper

US 65 and US 74.1

The issues with pair have been split off into a separate paper, which will be a revision of N2908.

Document Conventions

Although this paper does not propose formal wording, any reference to section names and numbers are relative to the **pre-concepts, August 2008** WP, N2723 (pre-San Francisco).

> Existing and proposed working paper text is indented and shown in dark blue. Small edits to the working paper are shown with ~~red strikeouts for deleted text~~ and <u>green underlining for inserted text</u> 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

Major Simplifications

The formal proposal for Santa Cruz will contain most or all of the following simplifications from the pre-concepts working paper (N2723):

1. Eliminate the following trait class templates:
   `is_scoped_allocator`,
   `constructible_with_allocator_prefix`,
   `constructible_with_allocator_suffix`,
   `allocator_propagate_never`,
   `allocator_propagate_on_copy_construction`,
   `allocator_propagate_on_move_assignment`,
   `allocator_propagate_on_copy_assignment`,
   `allocator_parpagation_map`

2. Remove specialization of `uses_allocator` for `pair` and `tuple`.

3. Eliminate the `construct_element` function template

4. Rename the second specialization of `scoped_allocator_adaptor` to `scoped_allocator_adapator2` to avoid confusion between the two usages.

5. Move `ConstructibleAsElement` requirements from [container.requirements] to the scoped-allocator section [allocator.adaptor].

6. Precisely specify requirements for a C++0x allocator.

7. Create an `allocator_traits` template that can be used to adapt other types to the allocator requirements. The `allocator_traits` template would be specialized in such a way that legacy C++03 allocators would automatically be so adapted.

Most of the above items are either self-explanatory, or can be understood in rough outline by reading N2768 and N2840. I will elaborate on the last two items below and save the details of the other items for the formal proposal before Santa Cruz.

C++0x Allocator Requirements

The requirements for a C++0x allocator can be specified either as a requirements table, or in a concept-like format. I will use the concept-like format here for brevity, but this will probably need to be converted to a requirements table for Santa Cruz.

```
struct Allocator
{
    typedef object-type value_type;

    typedef pointer-like-type pointer;
    typedef pointer-like-type const_pointer;

    typedef pointer-like-type generic_pointer;
    typedef pointer-like-type const_generic_pointer;

    typedef integer-type difference_type;
    typedef integer-type size_type;

    template <typename T> using rebind_type = rebind-template;

    // Static functions
    static pointer from_generic_pointer(generic_pointer);
    static const_pointer from_generic_pointer(const_generic_pointer);

    // Required constructor
    template <typename T>
      Allocator(const rebind_type<T>& other);
```

```
    // Allocator propagation on construction
    static Alloc select_on_copy_construction(const Alloc& rhs);

    // Allocator propagation functions.  Return true if *this was modified.
    bool do_on_container_copy_assignment(const Allocator& rhs);
    bool do_on_container_move_assignment(Allocator&& rhs);
    bool do_on_container_swap(Allocator& other);

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

    void deallocate(pointer p, size_type n);

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

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

    size_type max_size() const;

    pointer address(value_type& r) const;
    const_pointer address(const value_type& r) const;
};

bool operator==(const Allocator& a, const Allocator& b);
bool operator!=(const Allocator& a, const Allocator& b);
```

The `allocator_traits` class template

Instead of directly using an allocator *a* of type, *Alloc*, a client (i.e., a container) would access the allocator via the `allocator_traits` template:

```
typedef allocator_traits<Alloc> atraits;
p = atraits::allocate(a, 1);
```

This traits approach is clean and non-intrusive. It provides an adaptation point whereby almost any type can be used as an allocator (similar to the way `iterator_traits` allows pointers to be used as iterators). In addition, the traits approach is extensible because it provides a place for adding default implementations of new features in the future. Some meta-programming will be used to select an adapted traits specialization for legacy (C++03) allocators, probably by detecting the absence of `generic_pointer` and or `rebind_type`, or by using some kind of versioning as described in Howard Hinnant's paper, N1953. The `allocator_traits` template will look something like the following:

```
template <typename Alloc>
struct allocator_traits
```

```
{
    typedef Alloc allocator_type;

    typedef typename Alloc::value_type           value_type;

    typedef typename Alloc::pointer              pointer;
    typedef typename Alloc::const_pointer        const_pointer;

    typedef typename Alloc::generic_pointer      generic_pointer;
    typedef typename Alloc::const_generic_pointer const_generic_pointer;

    typedef typename Alloc::difference_type      difference_type;
    typedef typename Alloc::size_type            size_type;

    template <typename T> using rebind_type =
      Alloc::template rebind_type<T>;

    // Static functions
    static pointer from_generic_pointer(generic_pointer p)
        { return Alloc::from_generic_pointer(p); }
    static const_pointer from_generic_pointer(const_generic_pointer p)
        { return Alloc::from_generic_pointer(p); }

    // Allocator propagation on construction
    static Alloc select_on_copy_construction(const Alloc& from);

    // Allocator propagation on assignment and swap.
    // Return true if lhs is modified.
    static bool do_on_container_copy_assignment(Alloc& lhs,
                                                const Alloc& rhs)
        { return lhs.do_on_container_copy_assignment(rhs); }
    static bool do_on_container_move_assignment(Alloc& lhs, Alloc&& rhs)
        { return lhs.do_on_container_move_assignment(rhs); }
    static bool do_on_container_swap(Alloc& lhs, Alloc& rhs)
        { return lhs.do_on_container_swap(rhs); }

    static pointer allocate(Alloc& a, size_type n)
        { return a.allocate(n); }
    static pointer allocate(Alloc& a, size_type n,
                            const_generic_pointer hint)
        { return a.allocate(n, hint); }

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

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

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

```
        { a.destroy(p); }

    static size_type max_size(const Alloc& a, )
        { return a.max_size(); }

    static pointer address(const Alloc& a, value_type& r)
        { return a.address(r); }
    static const_pointer address(const Alloc& a, const value_type& r)
        { return a.address(r); }
};
```

What is the best way to detect a C++98 allocator for specializing `allocator_traits`?

Alternative: an adaptor for C++98 Allocators

In my sample implementation, I found myself implementing something like `legacy_allocator_adaptor`, below that combined the traits and the allocator into a single object. This adaptor would eliminate the need for `allocator_traits`, but we would still need a selection mechanism for C++0x vs. C++03 allocators:

`template <typename Alloc> using select_allocator_type =` *some-type*;

Where *some-type* is either `Alloc` (for a C++0x allocator) or `legacy_allocator_adaptor<Alloc>` (for legacy C++03 allocators). The selection criteria will probably be the existence in `Alloc` of `rebind_type` and/or `generic_pointer`, or by using some kind of versioning as described in Howard Hinnant's paper, N1953.

Does anybody on the committee feel strongly about the merits `allocator_traits` vs. `legacy_allocator_adaptor` mechanism?

```
template <typename Alloc>
struct legacy_allocator_adaptor
{
    typedef Alloc legacy_allocator_type;
    Alloc alloc_;   // exposition only

    typedef typename Alloc::value_type      value_type;

    typedef typename Alloc::pointer         pointer;
    typedef typename Alloc::const_pointer   const_pointer;

    typedef typename
      legacy_generic_pointer<pointer>       generic_pointer;
    typedef typename
      legacy_generic_pointer<const_pointer> const_generic_pointer;

    typedef typename Alloc::difference_type difference_type;
    typedef typename Alloc::size_type       size_type;
```

```
template <typename T> using rebind_type =
    legacy_allocator_adaptor<Alloc::template rebind<T>::other>;

// Static functions
static pointer from_generic_pointer(generic_pointer p)
    { return Alloc::from_generic_pointer(p); }
static const_pointer from_generic_pointer(const_generic_pointer p)
    { return Alloc::from_generic_pointer(p); }

// Constructors
template <typename T>
  legacy_allocator_adaptor(const rebind_type<T>& other)
      : alloc_(other.alloc_) { }

template <typename... Args>
legacy_allocator_adaptor(Args&&... args)
    : alloc_(std::forward<Args>(args)...) { }

// Allocator propagation constructor
legacy_allocator_adaptor(on_container_move_t,
                         const legacy_allocator_adaptor& other)
    : alloc_(other.alloc_) { }

// Allocator propagation functions. Return true if *this was modified.
bool
do_on_container_copy_assignment(const legacy_allocator_adaptor& from)
    { return false; }
bool do_on_container_move_assignment(legacy_allocator_adaptor&& from)
    { return false; }
bool do_on_container_swap(legacy_allocator_adaptor& other)
    { assert(alloc_ == other.alloc_); return false; }

pointer allocate(size_type n)
    { return alloc_.allocate(n); }
pointer allocate(size_type n, const_pointer hint)
    { return alloc_.allocate(n, hint); }
pointer allocate(size_type n, const_generic_pointer hint)
    { return alloc_.allocate(n); } // ignore hint

void deallocate(pointer p, size_type n)
    { alloc_.deallocate(p, n); }

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

// Special case maching legacy construct() signature
void construct(pointer p, const value_type& v)
    { alloc_.construct(p, v); }
```

```
    template <typename T>
      void destroy(T* p)
        { p->~T(); }

    size_type max_size() const
        { return alloc_.max_size(); }

    pointer address(value_type& r) const
        { return alloc_.address(r); }
    const_pointer address(const value_type& r) const
        { return alloc_.address(r); }
};

template <typename Alloc>
  bool operator==(const legacy_allocator_adaptor<Alloc>& a,
                  const legacy_allocator_adaptor<Alloc>& b);
template <typename Alloc>
  bool operator!=(const legacy_allocator_adaptor<Alloc>& a,
                  const legacy_allocator_adaptor<Alloc>& b);
```

Conclusion

A number of the allocator complexities described in US 65 and US 74.1 can be dealt with by using concept-like thinking and moving more of the machinery into the scoped allocator adaptors and out of the rest of the library. A future paper, in time for the Santa Cruz mailing will add formal wording to the ideas described in this paper, and will incorporate any guidance I get from committee members before then.

References

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

N2768: Allocator Concepts, part 1 (revision 2)

N2554: The scoped allocator model (Rev 2)

N2525: Allocator-specific move and swap

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

N2840: Defects and Proposed Resolutions for Allocator Concepts (Rev 2)

N2908: Several Proposals to Simplify pair (Rev 1)