Doc No: N2525=08-0035
Date: 2008-02-04
Author: Pablo Halpern
Bloomberg, L.P.

phalpern@halpernwightsoftware.com

Allocator-specific Swap and Move Behavior

Contents

Please Read: Relationship to N2524

This proposal and N2524 *Conservative Swap and Move with Stateful Allocators* are mutually-exclusive approaches to the same problem. These proposals should be considered together and not more than one of them should move forward. These proposals represent my best efforts so far to to remove the weasel wording that currently allows an implementation to assume that all allocators of a given type to compare equal. Removing this weasel wording is essential for making stateful allocators useful in a portable way.

Motivation

LWG 431 and N1599 point out that by the current definition of `swap` for containers, two containers of the same type can always be swapped in constant time and with no exception thrown. However, if the two containers contain stateful allocators and if those allocators do not compare equal, a question arises as to what `swap` should do. Should it swap the allocators, do a linear-time swap of the contents of the containers, or should it be undefined behavior?

The situation is complicated by the advent of the move-assignment operation. Again, one would expect move-assignment to be a constant-time operation if the allocator of the object being overwritten is equal to the address of the object being moved. But what if the allocators are not equal? Should an allocator be required to be move-assignable and should the allocator be moved with its container, or should move-assignment be a linear-time?

Section 23.1, paragraph 8 of the C++98 standard states that a standard container uses the same allocator throughout its lifetime. Section 23.1, paragraph 4, Note A and 23.1 paragraph 10 of the C++98 standard states that swap for standard containers has constant complexity (O(1)) and doesn't throw an exception unless the comparison object throws an exception. The requirement that the same allocator be used throughout the lifetime of an object would rule out swapping or moving allocators. The requirement that a container' swap always be constant-time and not throw would rule out doing a linear-time swap of the contents. The only remaining option for swap, using process of elimination and maintaining backward compatibility, would be that swapping container with unequal allocators must be undefined behavior. The last is the position taken in N2524.

In this paper, however, I propose that the behaviors of swap and move should be dependent on the intended usage pattern for the allocator and that the usage pattern is itself dependent on qualities of the allocator itself. The author of an allocator would decide on the properties of the allocator and define a trait (or concept map) that conveys those properties to the implementation of the containers. Backwards compatibility with C++98 is maintained by using the appropriate defaults so that new behavior is manafest only when using a newly-created allocator.

Basic Theory – allocator usage patterns

Let me begin by describing a spectrum of different allocator usage patterns. In this discussion, I use the term, "allocator mechanism", to mean the possibly-shared resource managed by the allocator. The role of an allocator mechanism would be a combination of roles of the "Allocator Storage" and "Allocator Manager" in Lance Diduck's paper, N2486. The term, "allocator", plays the role of the "Allocator Adaptor" in N2486. This mapping of terms to N2486 refers to roles only, not to interfaces. The extremes of allocator usage patterns are:

Pattern A) The allocator instance is a configuration parameter of an object, indicating where memory for this object should come from. The allocator should not change during the object's lifetime. The allocator mechanism is sometimes created in the same scope as the client object, requiring that the allocator

not be copied into an object that will outlive that scope (e.g. by copy-construction of the client into a longer-lived object).

Pattern B) The allocator is used for tuning the performance of a (container) class. The type of the allocator, not the instance, determines the characteristics. For this reason, one doesn't care so much what allocator instance a container is using, but there are performance benefits to having multiple stateful allocators rather than having one big memory pool shared by any number of stateless allocators. Otherwise, the allocators are more-or-less equivalent, though they do not compare equal because they manage different segments of memory. With this pattern, it is both safe and correct to change the allocator instance during the lifetime of an object whenever efficiency dictates that approach. The underlying memory resources used by this type of allocator are either long-lived or reference-counted so that one does not worry excessively about an object outliving the memory resource.

Both patterns are valid and there are some in-between varieties that I'll get to in a moment.

Examples: An allocator that gets memory from a fixed-sized buffer that may reside on the stack will likely follow pattern A. A pooling allocator that aims for locality of reference might follow pattern B. A shared memory allocator might follow either category, or one of the other varieties, depending on the purpose of the shared memory.

With this descriptive framework, I propose that the correct behavior for swap() as well as for copy-construction, move-construction, move-assignment, and copy-assignment for containers depends on what kind of allocator you are using. The proposal identifies four allocator usage patterns identified by traits or concepts. Each pattern propagates the allocator under more circumstances. The default trait is chosen to preserve C++98 expectations.

Summary of Changes to the Working Paper

I am proposing four traits or concepts to control when and how an allocator gets propagated during copy, move, and swap operations. For a c1 and c2 belonging to container type C with allocator_type A, the traits define the behavior as follows:

propagate_never

Operation	Postconditions	Complexity
C c2(c1)	c2.get_allocator() == A()	linear time
C c2(rv)	c2.get_allocator() == A()	linear time

c2 = rv	c2.get_allocator() unchanged	constant-time if c1.get_allocator() == c2.get_allocator(), else linear time
c2 = c1	c2.get_allocator() is unchanged	linear time
c2.swap(c1)	c2.get_allocator() is unchanged c1.get_allocator() is unchanged	constant time. Behavior is undefined unless c1.get_allocator() == c2.get_allocator()

propagate_on_copy_construction

Operation	Postconditions	Complexity
C c2(c1)	c2.get_allocator() == c1.get_allocator()	linear time
C c2(rv)	c2.get_allocator() == c1.get_allocator()	constant time
c2 = rv	c2.get_allocator() unchanged	constant-time if c1.get_allocator() == c2.get_allocator(), else linear time
c2 = c1	c2.get_allocator() is unchanged	linear time
c2.swap(c1)	c2.get_allocator() is unchanged c1.get_allocator() is unchanged	constant time. Behavior is undefined unless c1.get_allocator() == c2.get_allocator()

propagate_on_move_assignment

Operation	Postconditions	Complexity
C c2(c1)	c2.get_allocator() == c1.get_allocator()	linear time
C c2(rv)	c2.get_allocator() == c1.get_allocator()	constant time
c2 = rv	c2.get_allocator() == c1.get_allocator()	constant time
c2 = c1	c2.get_allocator() is unchanged	linear time
c2.swap(c1)	c2.get_allocator() is swapped with c1.get_allocator()	constant time

propagate_on_copy_assignment

Operation	Postconditions	Complexity
C c2(c1)	c2.get_allocator() == c1.get_allocator()	linear time
C c2(rv)	c2.get_allocator() == c1.get_allocator()	constant time
c2 = rv	c2.get_allocator() == c1.get_allocator()	constant time
c2 = c1	c2.get_allocator() == c1.get_allocator()	linear time
c2.swap(c1)	c2.get_allocator() is swapped with c1.get_allocator()	constant time

Document Conventions

All section names and numbers are relative to the October 2007 working paper, N2461.

Existing and proposed working paper text is indented and shown in dark blue. Small edits to the working paper are shown with ~~green 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).

As of this writing, concepts have not yet been accepted into the working draft. Accordingly, the proposed wording does not use concepts. Alternative working-paper text that declares concepts and concept maps is enclosed in a red box. Even once concepts are accepted, it is hoped that the non-concept interface could define a de-facto standard method of implementing the elements of this proposal using a C++03 compiler.

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

Allocator usage pattern traits

In section 20.6 [memory], add the following definitions

```
template <typename Alloc>
  struct allocator_propagate_never;
template <typename Alloc>
  struct allocator_propagate_on_copy_construction;
template <typename Alloc>
  struct allocator_propagate_on_move_assignment;
template <typename Alloc>
  struct allocator_propagate_on_copy_assignment;
template <typename Alloc>
  struct allocator_propagation_map;
```

It has been proposed that the behavior of swap be regulated based on whether the allocator itself was swappable. I considered this but rejected it as insufficient for two reasons: 1) any allocator that doesn't declare a private or delete assignment operator is swappable, creating a dangerous default and 2) Swappable doesn't address issues of move assignment and does not allow for propagate_never functionality.

Before section 20.6.1, add the following subsection:

20.6.x Allocator propagation traits [alloc.propagation.traits]

```
template <typename Alloc>
  struct allocator_propagate_never : false_type { };
```

requires: Alloc is an Allocator

remark: if specialized to derive from `true_type` for specific allocator type, 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.

```
template <typename Alloc>
  struct allocator_propagate_on_copy_construction : see below
```

requires: Alloc is an Allocator

remark: if specialized to derive from `true_type` for specific allocator type, 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`.

```
template <typename Alloc>
  struct allocator_propagate_on_move_assignment : false_type { };
```

requires: Alloc is an Allocator

remark: if specialized to derive from `true_type` for specific allocator type, 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 { };
```

requires: Alloc is an Allocator

remark: if specialized to derive from `true_type` for specific allocator type, 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.

20.6.y allocator propagation map [alloc.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);
};
```

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

Remark: The `allocator_propagation_map` provides functions to be used by containers for manipulating the allocators during construction, assignment, and swap operations. The implementations of functions above are dependent on the allocator propagation traits of the specific Alloc.

20.6.y allocator propagation map members [alloc.propagation.members]

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

returns: if `allocator_propagate_never<Alloc>`, 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>`, assign to = forward(from), otherwise do nothing.

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

effects: if `allocator_propagate_on_copy_assignment<Alloc>`, 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>`, exchange the values of *a* and *b*. Otherwise, if a == b, do nothing. Otherwise the behavior is undefined.

```
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);
}

concept AllocatorPropagateNever<typename Alloc>
  : AllocatorPropagation<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) { … }
}

concept AllocatorPropagateOnCopyConstruction<typename Alloc>
  : AllocatorPropagation<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) { … }
}

concept AllocatorPropagateOnMoveAssignment<typename Alloc>
  : AllocatorPropagation<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) { … }
}

concept AllocatorPropagateOnCopyAssignment<typename Alloc>
  : AllocatorPropagation<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) { … }
}

template <Allocator Alloc>
    requires ! AllocatorPropagateNever<Alloc> &&
             ! AllocatorPropagateOnMoveAssignment<Alloc> &&
             ! AllocatorPropagateOnCopyAssignment<Alloc>
```

```
  concept_map AllocatorPropagateOnCopyConstruction<Alloc> {
}
```

Container Requirements

In section 23.1 [container.requirements] modify paragraph 8 as follows:

> Copy and move constructors for all container types defined in this clause ~~copy~~obtain an allocator ~~argument~~ from calling allocator_propagation_map<allocator_type>::select_for_copy_construction() on their respective first parameters. All other constructors for these container types take an Allocator& argument (20.1.2), an allocator whose value type is the same as the container's value type. A copy of this argument is used for any memory allocation performed, by these constructors and by all member functions, during the lifetime of each container object or until the allocator is replaced. The allocator may be replaced only via assignment or swap(). Allocator replacement is performed by calling allocator_propagation_map<allocator_type>::move_assign(),allocator_propagation_map<allocator_type>::copy_assign(), or allocator_propagation_map<allocator_type>::swap() within the implementation of the corresponding container operation. In all container types defined in this clause, the member get_allocator() returns a copy of the Allocator object used to construct the container, or to replace the allocator.~~256)~~

Remove footnote 256:

> ~~256) As specified in 20.1.2, paragraphs 4-5, the semantics described in this clause applies only to the case where allocators compare equal.~~

In section 23.1, table 87, change the selected row as follows:

`X u(rv);` `X u = rv;`		post: u shall be equal to the value that rv had before this construction Equivalent to: X u; u = rv;	~~constant~~ (Note B)
`a = rv;`	`X&`	post: a shall be equal to the value that rv had before this construction	~~constant~~ (Note C)

Modify the paragraph immediately following Table 87 as follows:

> Notes: the algorithms swap(), equal() and lexicographical_compare() are defined in clause 25. Those entries marked "(Note A)" should have constant complexity. Those entries marked "(Note B)" have constant complexity unless allocator_propagate_never<X::allocator_type> is true, in which case they have linear complexity. Those entries marked "(Note C)" have constant complexity if a.get_allocator() == rv.get_allocator() or if either allocator_propagate_on_move_assignment<X::allocator_type> or allocator_propagate_on_copy_assignment<X::allocator_type> is true and linear complexity otherwise.

Splice

In section 23.2.3.4 [list.opts], modify paragraph 2 as follows:

list provides three splice operations that destructively move elements from one list to another. The behavior of splice operations is undefined if `this->get_allocator() != x.get_allocator()`.

The reason people want splice is to do an O(1) nothrow movement of nodes from one list to another. The purpose would be defeated if this guarantee did not hold. Someone wishing to copy elements between lists with unequal allocators can use the existing insert() and erase() members. Adding explicit wording here removes another barrier to eliminating the weasel words in the standard.

Remove Weasel-wording regarding stateful allocators

In section [allocator.requirements] (20.1.2), modify the last paragraphs 4 and 5:

Implementations of containers described in this International Standard are permitted to assume that their Allocator template parameter meets the following ~~two~~ additional requirement~~s~~ beyond those in Table 40.

~~— All instances of a given allocator type are required to be interchangeable and always compare equal to each other.~~

— The typedef members pointer, const_pointer, size_type, and difference_type are required to be T*, T const*, std::size_t, and std::ptrdiff_t, respectively.

Implementors are encouraged to supply libraries that can accept allocators that encapsulate more general memory models ~~and that support non-equal instances.~~ In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in Table 40~~,~~ ~~and the semantics of containers and algorithms when allocator instances compare non-equal,~~ are implementation-defined.

Implementation Experience

The elements of this proposal have been implemented and tested within an open-source library. Allocators that use the equivalent of the `allocator_propagate_never` trait have been in use at Bloomberg LP for several years and allow us to reason precisely about the use of memory resources.