Doc No: N2436=07-0306
Date: 2007-10-05
Author: Pablo Halpern
Bloomberg, L.P.

phalpern@halpernwightsoftware.com

Small Allocator Fix-ups

Contents

Introduction

This document comprises the small subset of n2387, *Omnibus Allocator Fix-up Proposals*, that was approved by the Library Working Group during the morning session of October 3, 2007 in Kona. The rest of n2387 is still on the table, neither approved nor rejected by the LWG.

The following issues are resolved by this proposal:

- LWG 401: incorrect type casts in table 32 in lib.allocator.requirements
- LWG 634: turn address into boost::addressof
- Variadic construct member function (per n2268).
- LWG 258: Missing allocator requirement (transitive ==)

Document Conventions

All section names and numbers are relative to the August 2007 working draft, N2369.

> 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).

Comments and rationale mixed in with the proposed wording appears as shaded text.

Requests for LWG opinions and guidance appears with light (yellow) shading.

1. Modifications to `construct` and `destroy`

Motivation

Unless and until we decide otherwise, `pointer` is not required to be a raw pointer and is not to be a reasonable argument to placement `new`. Thus, as LWG 401 points out, it is inappropriate to define the meaning of the `construct` member function in terms of placement new.

Separately, N2268, Placement Insert for Containers, which was accepted into the WP in Toronto in July 2007, creates the requirement for a `construct` function that takes a variadic argument list.

Proposed Wording

In [allocator.requirements] (20.1.2), add the following row to Table 39:

`Args`	a template parameter pack
`args`	a function parameter pack with the pattern `Args&&`

In section [allocator.requirements] (20.1.2), table 40, change the rows that describe `construct` and `destroy` as follows.

~~`a.construct(p,t)`~~	~~(not used)~~	~~Effect: ::new((void*)p) T(t)~~
~~`a.construct(p,v)`~~	~~(not used)~~	~~Effect: ::new((void*)p) T(std::forward<V>(v))~~
`a.construct(p,args)`	(not used)	Effect: Constructs an object of type T at p by invoking `T(forward<Args>(args)…)`
`a.destroy(p)`	(not used)	Effect: ~~((T*)p)->~T()~~Destroys the object at p.

2. Clarification of `address` for the Default Allocator

Motivation

LWG 634 points out that the definition of `allocator<T>::address()` is broken if `T` has an overloaded `operator&()`.

Proposed Wording

In section [allocator.members] (20.6.1.1), modify the first two paragraphs as follows:

20.6.1.1 allocator members [allocator.members]

```
pointer address(reference x) const;
```

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

```
const_pointer address(const_reference x) const;
```

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

The above change addresses LWG 634 using nearly the exact wording in the proposed resolution. It ensures that `std::allocator<T>::address()` does the right thing if `operator&` is overloaded for `T`. Note that this definition of `address` applies only to the default allocator (though it makes sense for any allocator for which `pointer` is the same as `value_type*`).

3. Consistent Copy and Equality Semantics for Allocators

Motivation

As per **LWG 258** [allocator.requirements] 20.1.2, table 40 requires that two allocators of the same type compare equal if memory allocated through one allocator can be deallocated through the other. It also states that if X and Y are corresponding allocators for different types, T and U, and if a is of type X and b is of type Y, then X a(b) will yield the post-condition that Y(a) == b. In other words, conversion is reversible. This comes close to, but does not fully state, that operator== for allocators must be transitive, symmetric, and reflexive, and that Y(a) == Y(a).

As intuitive as these relationships may seem to some, there are reasoned opinions that these requirements are not needed and that there are useful allocators that could be built if these requirements were not present. For example, a small arena allocator that contains an array of bytes right within its footprint would not be equal even to a copy of itself. Never the less, I propose that operator== be transitive, symmetric, and reflexive (i.e., a proper equivalence relationship) and that copy-construction and conversion-construction imply that the copy compares equal to the original. The reasons are as follows:

1. It violates a principle of operator overloading that an operator have semantics vastly different from the standard meaning. For example, `operator+` should not mean multiplication.

2. Similarly, it is not reasonable to assume that copy-constructing an object will yield an object that does not compare equal to the original.

3. Many containers have already been written that make the standard assumptions about copy construction and `operator==`.

4. Some uses of allocators, such as type erasure or footprint optimizations require that an allocator be able to allocate a copy of itself. At least one implementation of `vector` that I've seen puts the allocator on the heap.

A stateful allocator in this proposal would be required to share state with all of its copies (including copy-conversions). However, the benefits of having an allocator with truly unique state can be obtained by using an allocator with shared state and bundling the state object with the container that uses the allocator.

Proposed Wording

In section [allocator.requirements] (20.1.2), add the following to Table 40:

`a1 == a2`	`bool`	returns true iff storage allocated from each can be deallocated via the other. operator== is reflexive, symmetric and transitive.
`a1 != a2`	`bool`	same as `!(a1 == a2)`
`X()`		creates a default instance [*Note:* destructor s assumed. – *end note*]
`X a1(a);`		post: `a1 == a`
`X a(b);`		post: `Y(a) == b` post: `a == X(b)`

These changes make copy-construction, comparison, and equality consistent with one another and with the common understanding of how they work.