Doc No: N2524=08-0034
Date: 2008-02-04
Author: Pablo Halpern
Bloomberg, L.P.

phalpern@halpernwightsoftware.com

Conservative Swap and Move with Stateful Allocators

Contents

Please Read: Relationship to N2525

This proposal and N2525 *Allocator-specific Swap and Move Behavior* 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?

Summary of Changes to the Working Paper

After much deep thought, many discussions, and countless sleepless nights, I propose that swapping containers with unequal allocators should yield undefined behavior. The rationale is as follows:

Let's start with the purpose of swap. Many people believe that swap should be a very fast (constant-time) operation and that one should be able to determine at coding time that it will not throw an exception. In fact, it has been asserted that swap is a fundamental value-type operation, as fundamental and important as copy construction or assignment. The known common uses of swap are:

a) In user code as a convenient way to swap values (rare)

b) In user code for the famous "swap trick" for creating exception-safe code by modifying a copy of an object, then, only when the modification is successful, quickly and safely swapping the modified value with the original.

c) In library code to permute elements in a sequence, i.e., sort(), rotate(), erase(), etc.

Usage (a) is the least sensitive to performance and exception guarantees, but is also the rarest. The other two cases would suffer if certain guarantees were not present.

Now let's consider the options. There are three front-running candidates for the meaning of swap for containers with unequal allocators. (For the moment, I'll omit discussion of selecting a behavior at compile time from among two alternatives based on a trait or concept. In order for that to be useful, *both* behaviors in in question must be useful.):

==========

OPTION (1) swap the allocators if they are unequal (or always swap them).

OPTION (2) never swap the allocator: O(n) swap if unequal allocators are involved.

OPTION (3) undefined behavior if the allocators are unequal.

OPTION (4) the behavior varies based on some trait(s)

==========

Each of these three alternatives will result in some existing code breakage. Given object X using allocator x and object Y using allocator y, some examples of breakage for swap(x, y) are:

Breakage with option (1) swap the allocators:

- violates the guarantee given in Section 23.1, paragraph 8 of the C++98 standard, which states that a container uses the same allocator throughout its lifetime
- crash if object X outlives the resource managed by allocator y
- performance bug if allocator x is tuned to the expected usage of X
- out-of-memory condition when object X is resized, but allocator y uses a limited memory resource
- crash if X is shared by multiple threads, but allocator y is intended to be tied to a specific thread (i.e., not thread safe). Note that this is an important use case for performance-critical software.

Breakage with option (2) O(n) swap:

- violates the guarantees given in section 23.1, paragraph 4, Note A and 23.1 paragraph 10 of the C++98 standard, which states that swap has constant complexity (O(1)) and doesn't throw an exception unless the comparison object throws an exception.
- latent "performance bugs" whereby a program will not exhibit problems until the data set gets large
- corrupt data structures (i.e., violation of class invariants) if swap() throws an exception in code that expects swap() to be nothrow.

Breakage with option (3) undefined behavior:

- assert failure likely in a debug build (and an extra 'if' statement)
- crash in optimized build (but zero overhead for equal allocators)

Breakage with option (4) (see N2525):

- all of the above, but under user control

The breakage with options (1), (2), and sometimes (4) are subtle and may elude testing in many cases. Once detected, these bugs may be very difficult to track down. I assert that it is much better to have the breakage be detectable rather than silent. Only option (3) provides the restricted preconditions needed to enable an efficient *optional* "safe" (debug) mode whereby the bug is easily and cheaply detected at the point of failure (or not-detected with zero overhead for optimized builds).

Even if option (3) is not the best choice in the long term, it is clearly correct in the short term for the following reasons:

- There is no consensus as to which of the other options should prevail, and so we need time to gain experience with stateful allocators. Hence, by chosing undefined behavior we keep our options open, thus allowing us to choose a different option in the future changing defined behavior.

- Moreover, even if we were 100% sure that one of the other options was the long-term correct behavior, we would still choose option (3) in the short term to allow a migration path that detects code that would silently break under the newly defined (future) behavior. After a sufficient interval (years?) we would adopt this defined behavior, and it would be fully backward compatible with preexisting code. The solution to breakage with all three options is to restructure your code so that the containers involved in swap() always have allocators that compare equal. Thus fixing code for option (3) paves the way for adopting options (1) or (2) or some other defined behavior in the future.

In conclusion, the undefined behavior option imposes zero overhead, is the easiest to implement, requires the fewest changes in the working paper, makes it by far the easiest to detect bugs, is consistent with a similar solution for list::splice, and gives us the most flexibility for future enhancements. What's not to love?

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

Container Requirements

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:

`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 C)" have constant complexity if a.get_allocator() == rv.get_allocator() and linear complexity otherwise.

This proposal retains the quality that an allocator does not change after construction of an object. The above change to the WP undoes an implicit relaxation of this rule. Given that move-assignment may occur in many places where copy-assignment previously occurred (e.g. a = f()), I felt it was imperative to fix this silent change in semantics.

Modify section 23.1, paragraph 10, 5th bullet as follows:

> — no swap() function throws an exception unless that exception is thrown by the copy constructor or assignment operator of the container's Compare object (if any; see 23.1.2). When swapping containers with unequal allocators, the behavior is undefined.

Short, sweet, and to the point.

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.