Allocators and swap

Respectng the contract of primary templates

Document #:	P0178R1
Date:	2024-12-17
Project:	Programming Language C++
Audience:	Library

Reply-to:	Alisdair Meredith <ameredith1@bloomberg.net>

Contents

1 Abstract

The free function template `swap` in namespace `std` has a wide contract. However, standard containers provide better matching overloads of `swap` in the same namespace with narrow contract. Substituting a function with another having a narrower contract is known to be problematic, especially in generic contexts where ADL-`swap` is often an implementation detail. This paper proposes widening the container `swap` functions by defining the currently undefined behavior to behave the same as a call to the primary `swap` template.

2 Revision history

R1 December 2024 (post-Wrocław mailing)

- Revised proposal reflecting changes of the last 9 years
  - standard dismantled container requirement tables
    - now free-function swap calls member swap
  - adopted proposal banning user specializations of `allocator_traits`, [P2652R2]
- Deliberately not touching node-handles due to lack of field experience
  - it looks like the same issue though

R0 February 2016 (pre-Jacksonville mailing)

Initial draft of this paper.

3 Introduction

The C++11 standard introduced `allocator_traits` as a way to easily support a variety of allocator models, especially those where allocators hold some kind of state that might affect allocator, such as a pointer to a memory resource, [N3916]. One of the problems that had to be tackled was the effect of calling `swap` on containers (or other allocator-aware types) that have allocators that do not compare equal. The committee at the time took the conservative option of simply making this undefined behavior, and a couple of issues have since been filed regarding some of the problems this causes: [LWG2152], [LWG2153]. The last time the Library Working Group looked at these issues, it called for a paper to better describe the issues and propose a solution (with wording!) This is that paper.

4 The Basic Problem

When support for stateful allocators was added to C++11, it became important to answer the question of what should happen when two containers with allocators that cannot allocate/deallocate memory on behalf of the other are swapped. The conservative position adopted at the time was to simply declare such calls out of contract, and make them undefined behavior. This gave us the maximum freedom to revise our position later, if necessary. It also matched the contract Bloomberg documented for their library, which was one of the code bases known to make widespread use of stateful allocators.

The problem with giving the free-function swap a narrow contract is that it does not satisfy the contract of the generic `swap` function in the same namespace. This greatly harms the ability to use `std::swap` in generic code, as the calling template must now be aware if the type it is instantiated for is a standard library container, and if so, whether its allocators compare equal. Conversely, this is not a problem for the `swap` member function, as the caller either knows exactly which container they are dealing with, or should document the container concept they are dealing with if it differs in some way from the standard. In some sense, the generic free function is more fundamental than the member function.

There has been some concern raised that the standard library cannot make arbitrary constraints on `swap` function found via ADL, so this exercise is vain. That is not entirely true, as if we assume the container requirements apply to user code, then we are already changing their `swap` contract by introducing undefined behavior. However, in practice, the current Container Requirements tables do not document Concepts, but are rather a form of common documentation to avoid redundant wording through each of the container sub-clauses. In this context, the standard library has broken its own `swap` function by adding an overload in its own namespace that, if not present, would allow the container to "do the right thing", and with move-semantics enabled for each container, do so efficiently with the correct exception-safety guarantees. The optimization offered by the overload, in a modern compiler, is tiny (but still worth taking, especially for containers with additional predicates).

The breakage of `swap` has more subtle implications too. The author of generic code could, by defining their own `is_standard_container` trait, rewrite their algorithm to allow for exchanging states of containers with unequal allocators. In fact, this is necessary for any generic algorithm that expects to work with containers unless the contract explicitly calls out this problem. The standard library offers no support for users trying to help their users in this manner. However, the further subtlety is that `is_standard_container` is not sufficient, as swap for pair and tuple is similarly undefined, and would need to be detected and supported separately. This problem further eats into any other type that wraps a standard container and can support stateful allocators.

5 Feedback from field experience

Bloomberg have over a decade of experience with a stateful allocator model in their standard library. When this was first introduced, the undefined behavior described by the current standard was added to all of our `swap` contracts, but the (undocumented) implementation performs the double-copy/swap described in this paper. This allowed existing code, written without stateful allocators in mind, to continue to function while the new contract was slowly adopted. A decade on, we still get large resistance any time we try to enable assertions on these narrow contracts, as perfectly sensible use-cases exist (frequently, not always, in generic code) for continuing to support this behavior. Given the choice between linear and potentially-throwing `swap` vs. undefined behavior, our users have been very clear which they prefer — as long as the constant-time, non-throwing behavior remains when the allocators do compare equal.

6 Proposed Solution

Due to the complexities of multiple traits affecting allocator propagation, absent adopting an update to [P0177R2] for force trait consistency, we must consider several scenarios:

- allocators compare as equal
  - allocators propagate on swap
  - allocators do not propagate on swap
- allocators compare not equal
  - allocators (consistently) always propagate
  - allocators (consistently) never propagate
  - allocators propagate on swap, but not on assignment
  - allocators propagate on assignment, but not on swap

As you can see, the easiest scenario is when the allocators compare as equal:

When allocators compare equal, we expect a container to have a fast-path to exchange internal handles to the dynamic data structure. That is a runtime property from some allocators, and the test can be skipped for allocators with the `allocators_always_compare_equal` allocator trait. Likewise, if the trait for allocator propagate on swap is `true`, allocators should be exchanged, even if equal, in case there is associated data in the allocator such as a name.

When allocators do not compare equal

```
void CONTAINER_TYPE::swap(CONTAINER_TYPE & other) {
   if( std::allocator_traits<allocator_type>::is_always_equal // compiler will optimize to compile-time
    || this->get_allocator() == other.get_allocator()) {
      // take the fast path

      if constexpr(std::allocator_traits<allocator_type>::propagate_on_container_swap ) {
         using std::swap;
         swap( this->get_allocator(), other.get_allocator());
      }
   }
   else ...
}
```

Now let us consider the case that allocators have a consistent set of propagation traits — all `true` or all `false`. In this case we can rely on assignment to propagate, or not, the allocator as needed, and the `swap` implementation remains the simple swap-through-a-temporary implementation.

That leaves the final concern, that [P0177R2] would rule out, where assignment and swap diverge on propagation. In this case, we create two buffers using the corresponding allocator of the other container. Once we have constructed both of these buffer containers, we have the guaranteed fast-path swap of two containers having the same allocator.

The last wrinkle is whether the allocators were supposed to propagate when not equal but do propagate on assignment — and you will notice that case should be covered by the fast path, so we can add that check there.

```
void CONTAINER_TYPE::swap(CONTAINER_TYPE & other) {
   if( std::allocator_traits<allocator_type>::propagate_on_container_swap
    || std::allocator_traits<allocator_type>::is_always_equal
    || this->get_allocator() == other.get_allocator()) {
      // trust compiler will optimize the compile-time branching

      // take the fast path

      if constexpr(std::allocator_traits<allocator_type>::propagate_on_container_swap) {
         using std::swap;
```

```
         swap( this->get_allocator(), other.get_allocator());
      }
   }
   else if constexpr( all traits are consistent ) {
      CONTAINER buffer{std::move(other)};
      other = std::move(*this);
      *this = std::move(buffer);
   }
   else {
      // Copy into buffer having the right allocator
      CONTAINER buffer_this {std::move(other), this->get_allocator()};
      CONTAINER buffer_other{std::move(*this), other.get_allocator()};

      this->swap(buffer_this);
      other.swap(buffer.other);
   }
}
```

The simplest implementation of the `swap` overload (for a given `CONTAINER_TYPE`) should be efficient and correct, without preconditions:

```
void swap(CONTAINER_TYPE & left, CONTAINER_TYPE & right) {
   std::swap<CONTAINER_TYPE>(left, right);
}
```

However, the optimal algorithm for swapping containers (or any other allocator-aware type) might be more like:

```
void swap(CONTAINER_TYPE & left, CONTAINER_TYPE & right) {
   if (allocators are compatible) {
      left.swap(right);
   }
   else if (allocator propagation traits are sane) {
      std::swap<TYPE>(left, right);
   }
   else {
      CONTAINER_TYPE tempLeft {std::move(right), left.get_allocator() };
      CONTAINER_TYPE tempRight{std::move(left ), right.get_allocator()};
      swap(left,  tempLeft );
      swap(right, tempRight);
   }
}
```

7 Wording

Make the following changes to the C++ Working Draft. All wording is relative to [N5001], the latest draft at the time of writing.

23.2.2.1 [container.intro.reqmts] Introduction

65 The expression `a.swap(b)`, for containers `a` and `b` of a standard container type other than `array` and `inplace_vector`, shall exchange the values of `a` and `b` without invoking any move, copy, or swap operations on the individual container elements. Any `Compare`, `Pred`, or `Hash` types belonging to `a` and `b` shall meet the *Cpp17Swappable* requirements and shall be exchanged by calling `swap` as described in 16.4.4.3 [swappable.requirements]. If `allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`, then `allocator_type` shall meet the *Cpp17Swappable* requirements and the allocators of `a` and `b` shall also be exchanged by calling `swap` as described in 16.4.4.3 [swappable.requirements]. Otherwise, the allocators shall not be swapped ~~, and the behavior is undefined unless `a.get_allocator() == b.get_allocator()`~~. Every iterator referring to an element in one container before the swap shall refer to the same element in the other container after the swap. It is unspecified whether an iterator with value `a.end()` before the swap will have value `b.end()` after the swap.

66 Unless otherwise specified (see 23.2.7.2 [associative.reqmts.except], 23.2.8.2 [unord.req.except], 23.3.5.4 [deque.modifiers], 23.3.14.5 [inplace.vector.modifiers], and 23.3.11.5 [vector.modifiers]) all container types defined in this Clause meet the following additional requirements:

(66.1) — If an exception is thrown by an `insert()` or `emplace()` function while inserting a single element, that function has no effects.

(66.2) — If an exception is thrown by a `push_back()`, `push_front()`, `emplace_back()`, or `emplace_front()` function, that function has no effects.

(66.3) — No `erase()`, `clear()`, `pop_back()` or `pop_front()` function throws an exception.

(66.4) — No copy constructor or assignment operator of a returned iterator throws an exception.

(66.5) — No `swap()` function throws an exception unless

(66.5.1) — `allocator_traits<allocator_type>::propagate_on_container_swap::value` is `false`, and

(66.5.2) — `a.get_allocator() != b.get_allocator()`.

(66.6) — No `swap()` function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped unless

(66.6.1) — `allocator_traits<allocator_type>::propagate_on_container_swap::value` is `false`, and

(66.6.2) — `a.get_allocator() != b.get_allocator()`.

[*Note 4:* The `end()` iterator does not refer to any element, so it can be invalidated. —*end note*]

23.2.2.2 [container.reqmts] Container requirements

```
t.swap(s)
```

48 *Result:* `void`.

49 *Effects:* Exchanges the contents of `t` and `s`.

50 *Complexity:* Linear for `array` and `inplace_vector`, and constant for all other standard containers unless

(50.1) — `allocator_traits<allocator_type>::propagate_on_container_swap::value` is `false`, and

(50.2) — `t.get_allocator() != s.get_allocator()`.

8 Acknowledgements

Thanks to Michael Park for the pandoc-based framework used to transform this document's source from Markdown.

9 References

[LWG2152] Robert Shearer. Instances of standard container types are not swappable.
https://wg21.link/lwg2152

[LWG2153] Robert Shearer. Narrowing of the non-member swap contract.
https://wg21.link/lwg2153

[N3916] Pablo Halpern. 2014-02-14. Polymorphic Memory Resources - r2.
https://wg21.link/n3916

[N5001] Thomas Köppe. 2024-12-17. Working Draft Programming Languages — C++.
https://wg21.link/n5001

[P0177R2] Alisdair Meredith. 2016-03-21. Cleaning up allocator_traits.
https://wg21.link/p0177r2

[P2652R2] Pablo Halpern. 2023-02-09. Disallow user specialization of allocator_traits.
https://wg21.link/p2652r2