P1843R0 Comparison and Hasher Requirements

Billy O'Neal <bion@microsoft.com>

Audience: LEWG

Discussion

This paper is in response to review of LWG 2189 on Wednesday night in Cologne 2019, in which LWG expressed a desire for option 2, and to fix the ordered containers as well.

For those unfamiliar with the problem, when we are implementing unordered_Xxx::swap, there isn't a reasonable way for an implementation to tolerate a throwing comparison or hasher function object. We can't even provide the basic exception safety guarantee – the container has become inconsistent and the hasher and equality predicate might no longer agree if one swap succeeds and the other does not or vice versa.

Given that there doesn't seem to be a reasonable use case for swap to throw in this condition, and MSVC++, libstdc++, and libc++ all fail to tolerate this condition, we should fix the spec to make it clear this need not be supported.

There's a similar problem in priority_queue where if we succeed to swap the containers but fail to swap the comparison function objects, the result is undefined behavior in the priority queue. We should require that the container preserves the priority_queue invariants during swap, and prohibit throwing from the comparison function object.

It is technically possible to support this in the ordered associative containers because there's only one functor to swap; if we fail to swap we can still provide at least the basic guarantee by clearing both containers.

LEWG Question: Given that we need to fix hash and equality for unordered associative containers, should we change the ordered versions for consistency even though they are implementable as specified?

Proposed Wording

This wording is relative to N4810, and assumes we are also fixing the ordered containers.

In [container.requirements.general]/9:

-9- The expression `a.swap(b)`, for containers `a` and `b` of a standard container type other than `array`, shall exchange the values of `a` and `b` without invoking any move, copy, or swap operations on the individual container elements. Lvalues of any `Compare`, `Pred`, or `Hash` types belonging to `a` and `b` shall be swappable and **are**~~**shall be**~~ exchanged by calling `swap` as described in 16.5.3.2. **The call to swap shall not exit via an exception.** If

allocator_traits<allocator_type>::propagate_on_container_swap::value is true , then lvalues of type allocator_type shall be swappable and the allocators of a and b shall also be exchanged by calling swap as described in 16.5.3.2. 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.

Delete [associative.reqmts.except]/3:

~~-3- For associative containers, no swap function throws an exception unless that exception is thrown by the swap of the container's Compare object (if any).~~

Delete [unord.req.except]/3:

~~-3- For unordered associative containers, no swap function throws an exception unless that exception is thrown by the swap of the container's Hash or Pred object (if any).~~

In the map synopsis [map.overview]/2, update the map member swap noexcept expression:

```
void swap(map&)
noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
allocator_traits<Allocator>::is_always_equal::value~~ &&
~~is_nothrow_swappable_v<Compare>~~);
```

And the multimap synopsis [multimap.overview]/2:

```
void swap(multimap&)
noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
allocator_traits<Allocator>::is_always_equal::value~~ &&
~~is_nothrow_swappable_v<Compare>~~);
```

And in [set.overview]/2:

```
void swap(set&)
noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
allocator_traits<Allocator>::is_always_equal::value~~ &&
~~is_nothrow_swappable_v<Compare>~~);
```

And in [multiset.overview]/2:

```
void swap(multiset&)
noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
allocator_traits<Allocator>::is_always_equal::value~~ &&
~~is_nothrow_swappable_v<Compare>~~);
```

And in [unord.map.overview]/3:

```
void swap(unordered_map&)
noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
```

```
allocator_traits<Allocator>::is_always_equal::value &&
is_nothrow_swappable_v<Hash> && is_nothrow_swappable_v<Pred>);
```

And in [unord.multimap.overview]/3:

```
void swap(unordered_multimap&)
noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
allocator_traits<Allocator>::is_always_equal::value &&
is_nothrow_swappable_v<Hash> && is_nothrow_swappable_v<Pred>);
```

And in [unord.set.overview]/3:

```
void swap(unordered_set&)
noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
allocator_traits<Allocator>::is_always_equal::value &&
is_nothrow_swappable_v<Hash> && is_nothrow_swappable_v<Pred>);
```

And in [unord.multiset.overview]/3:

```
void swap(unordered_multiset&)
noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
allocator_traits<Allocator>::is_always_equal::value &&
is_nothrow_swappable_v<Hash> && is_nothrow_swappable_v<Pred>);
```

In [priqueue.overview]/1, remove the inline swap and fix up the noexcept:

```
void swap(priority_queue& q) noexcept(is_nothrow_swappable_v<Container> &&
is_nothrow_swappable_v<Compare>)
{ using std::swap; swap(c, q.c); swap(comp, q.comp); }
```

In [priqueue.members], add:

`void swap(priority_queue& q) noexcept(is_nothrow_swappable_v<Container>)`

-?- *Constraints:* `is_swappable_v<Container>` is `true` and `is_swappable_v<Compare>` is `true`.

-?- *Expects*: If swapping `this->c` with `q.c` throws an exception, either there are no effects on the containers, or they both contain 0 elements. Swapping `this->comp` and `q.comp` shall not exit via an exception.

-?- *Effects:* Exchanges the contents of `*this` and `q` by: `using std::swap; swap(c, q.c); swap(comp, q.comp);`

In [priqueue.special], delegate to the member swap:

```
template<class T, class Container, class Compare>
void swap(priority_queue<T, Container, Compare>& x, priority_queue<T,
Container, Compare>& y) noexcept(noexcept(x.swap(y)));
```

~~-1- *Constraints:* `is_swappable_v<Container>` is `true` and `is_swappable_v<Compare>` is `true`.~~

-2- *Effects:* ~~As if by~~ **Equivalent to:** x.swap(y).

Thanks

Thanks to Daniel Krügler for his stewardship of all the LWG issues things.

Thanks to Stephan T. Lavavej for reviewing this proposal before submission of R0.