WG21 Document Number: P0208Rr0
Date: 2016-02-12
Intended audience: LEWG
Pablo Halpern <phalpern@halpernwightsoftware.com>

Copy-Swap Helper

Motivation

A favorite idiom for writing exception-safe code is to employ the *copy-swap idiom*. In general, the copy-swap idiom involves making a copy of an object and modifying the copy. Once the modification is successful and does not throw an exception the original object and the copy are swapped. If an exception is thrown during modification of the copy, however, the original object is left unchanged, providing what is often called *the strong guarantee* of exception safety. In pseudo-C++, the copy-swap idiom for safely modifying an object x of type T is:

```
T xprime(x);
// modify xprime here (might throw)
…
using std::swap;
swap(x, xprime);  // Does not throw
```

A variation of this idiom is commonly used to get the strong guarantee in the implementation of a copy-assignment operator:

```
T& T::operator=(const T& rhs)
{
    T(rhs).swap(*this); // T::swap does not throw
    return *this;
}
```

The problem with this idiom is that if T is an allocator-aware type, the allocator instance used for the copy might not be the correct allocator instance to use for the swap. In the assignment-operator example, if `rhs` has a different allocator than `*this`, it is likely that the temporary copy `T(rhs)` will have the same allocator as `rhs` and a different allocator than `*this`. Unless the allocator type has the `propagate_on_container_swap` trait set to true (a rarity), the swap becomes undefined behavior and is likely to fail, not with an exception, but with an assertion failure or worse.

The general copy-swap idiom for modifying a single object of type T is less likely to fail because most allocators do propagate on copy construction. Such propagation is not guaranteed, however, with `pmr::polymorphic_allocator` in the fundamentals TS being an example of an allocator that does not propagate on copy construction of the container.

Summary of proposal

This paper proposes two or three function templates that can be used to solve the problems above and have the added benefit of annotating the use of the copy-swap idiom in user code. The functions use metaprogramming to determine if a type uses an allocator and, if so, it ensures that the temporary copy used for the copy-swap idiom uses the correct allocator. Because the presence or absence of an allocator is determined at compile-time, these function templates are usable in generic code, where the type being swapped may or may not use an allocator. The general copy-swap idiom using these facilities would look like the following:

```
T xprime(copy_swap_helper(x));
// modify xprime here (might throw)
…
using std::swap;
swap(x, xprime);  // Does not throw
```

The assignment operator example would be rewritten as follows:

```
T& T::operator=(const T& rhs)
{
    copy_swap_helper(rhs, *this).swap(*this); // T::swap does not throw
    return *this;
}
```

Target publication

These functions can be targeted for C++17 or the third revision of the Library Fundamentals TS (LFTS-3) or both, as determined by the LEWG. It should be noted that the problem being solved has existed since C++11 and that the facility being proposed has been fully implemented.

Implementation experience

The functions described in this paper have been fully implemented and well tested. The code (including test driver) is available at https://github.com/phalpern/uses-allocator.

Alternative design

The two-argument `copy_swap_helper` performs the copy or move but not the actual swap. If the assignment idiom is the only use of these functions, it may be reasonable to change their name to `swap_assign` and have it do the entire options, thus simplifying the use of the idiom for assignment:

```
T& T::operator=(const T& rhs)
{
    return swap_assign(*this, rhs);
}
```

However, there may be a more general use for making a copy of an object using the allocator from a different object of the same type, so it was decided to keep the functionality separate. Another alternative is to offer both.

The name `copy_swap_helper` is specific to the idiom. However, the functionality of producing a copy of an object with an allocator from the same or a different object could have broader applicability. Names that convey the meaning *make-a-copy-using-the-same-allocator* and *make-a-copy-using-the-allocator-from-x* might be better choices, and I am willing to entertain such names.

Proposed Wording

Text that makes sense only in the LFTS is shaded grey. It would not be copied to C++17 unless and until polymorphic resources are moved to C++17.

Add the following feature test macro to section 1.6 [general.feature.test] of the LFTS:

Doc no.	Title	Primary Section	Macro Name Suffix	Value	Header
P0208	Copy-Swap Helper	TBD	`copy_swap_helper`	201602	`<experimental/memory>`

Add to header `<memory>` (or `<experimental/memory>`) synopsis:

```
namespace std {
namespace experimental {
inline namespace fundamentals_v3 {

template <class T>
  remove_reference_t<T> copy_swap_helper(T&& other);

template <class T, class U>
  remove_reference_t<T> copy_swap_helper(T&& other,
                                         const U& alloc_source);
```

The following is proposed instead of, or in addition to, the previous (two-argument) function templates.

```
template <class T>
  T& swap_assign(T& lhs, decay_t<T> const& rhs);
template <class T>
  T& swap_assign(T& lhs, decay_t<T>&& rhs);

}}
}
```

Add the following descriptions for the above function templates:

```
template <class T>
  remove_reference_t<T> copy_swap_helper(T&& other);
```

Effects: Defines a value R as follows:

— If `T` is an rvalue, then R is `std::move(other)`

— Otherwise, if `other.get_memory_resource()` is well formed and `uses_allocator_v<T, memory_resource*>` is true, then R is an object of type T constructed by *uses-allocator construction* ([allocator.uses.construction] in the C++ standard) with allocator `other.get_memory_resource()` and argument `std::forward<T>(other)`.

— Otherwise, if `other.get_allocator()` is well formed and `uses_allocator_v<T, decltype(other.get_allocator())>` is true, then R is an object of type T constructed by *uses-allocator construction* ([allocator.uses.construction] in the C++ standard) with allocator `other.get_allocator()` and argument `std::forward<T>(other)`.

— Otherwise, R is `std::forward<T>(other)`

Returns: The value R, as defined in the *effects* clause, above.

```
template <class T, class U>
  remove_reference_t<T> copy_swap_helper(T&& other,
                                         const U& alloc_source);
```

Effects: Defines a value R as follows:

— If `alloc_source.get_memory_resource()` is well formed and `uses_allocator_v<T, memory_resource*>` is true, then R is an object of type T constructed by *uses-allocator construction* ([allocator.uses.construction] in the C++ standard) with allocator `alloc_source.get_memory_resource()` and argument `std::forward<T>(other)`.

— Otherwise, if `alloc_source.get_allocator()` is well formed and `uses_allocator_v<T, decltype(alloc_source.get_allocator())>` is true, then R is an object of type T constructed by *uses-allocator construction* ([allocator.uses.construction] in the C++ standard) with allocator `alloc_source.get_allocator()` and argument `std::forward<T>(other)`.

— Otherwise, R is `std::forward<T>(other)`

Returns: The value R, as defined in the *effects* clause, above.

The following is proposed instead of, or in addition to, the previous (two-argument) function templates.

```
template <class T>
  T& swap_assign(T& lhs, decay_t<T> const& rhs);
```

Effects: `swap(lhs, R)`, where R is defined as follows:

— If `lhs.get_memory_resource()` is well formed and `uses_allocator_v<T, memory_resource*>` is true, then R is an object of type T constructed by *uses-allocator construction* ([allocator.uses.construction] in the C++ standard) with allocator `lhs.get_memory_resource()` and argument `rhs` for `copy_swap` or `std::move(rhs)` for move swap.

— Otherwise, if `lhs.get_allocator()` is well formed and `uses_allocator_v<T, decltype(lhs.get_allocator())>` is true,

- If `allocator_traits<decltype(lhs.get_allocator())>::propagate_on_container_copy_assignment::value` is `true`, then R is an object of type T constructed by *uses-allocator construction* ([allocator.uses.construction] in the C++ standard) with allocator `rhs.get_allocator()` and argument `std::forward<T>(rhs)`
- Otherwise R is an object of type T constructed by *uses-allocator construction* ([allocator.uses.construction] in the C++ standard) with allocator `lhs.get_allocator()` and argument `rhs`.

— Otherwise, R is `rhs`.

Returns: `lhs`

```
template <class T>
  T& swap_assign(T& lhs, decay_t<T>&& rhs);
```

Effects: `swap(lhs, R)`, where R is defined as follows:

— If `lhs.get_memory_resource()` is well formed and `uses_allocator_v<T, memory_resource*>` is true, then R is an object of type T constructed by *uses-allocator construction* ([allocator.uses.construction] in the C++ standard) with allocator `lhs.get_memory_resource()` and argument `rhs` for `copy_swap` or `std::move(rhs)` for move swap.

— Otherwise, if `lhs.get_allocator()` is well formed and `uses_allocator_v<T, decltype(lhs.get_allocator())>` is true,

- If `allocator_traits<decltype(lhs.get_allocator())>::propagate_on_container_move_assignment::value` is `true`, then R is `T(std::move(rhs))`.
- Otherwise R is an object of type T constructed by *uses-allocator construction* ([allocator.uses.construction] in the C++ standard) with allocator `lhs.get_allocator()` and argument `std::move(rhs)`.

— Otherwise, R is `rhs`.

Returns: `lhs`