WG21 Document Number: P0208r1
Date: 2017-06-17
Intended audience: LEWG
Pablo Halpern <phalpern@halpernwightsoftware.com>

Copy-Swap Transaction

Changes since R0

- Changed title to "Copy-Swap Transaction" from "Copy-Swap Helper."
- Proposes a transaction-like function, `copy_swap_transaction` instead of a factory function `copy_swap_helper`.
- Adds `get_allocator` function template.
- Removed formal wording that relates to `memory_resource*`. A better approach, described in P0339, eliminates the need for special handling of `memory_resource*`.

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:

```
try {
    T xprime(x);
    modify xprime here (might throw)
    …
    using std::swap;
    swap(x, xprime);  // Does not throw
} catch (etc.) { … }
```

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; `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 four 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:

```
try {
    std::copy_swap_transaction(x, [&](auto& xprime){
        modify xprime here (might throw)
        …
    });
} catch (etc.) { … }
```

Note that a single call to `copy_swap_transaction` may be used to modify multiple variables safely, as follows:

```
try {
    std::copy_swap_transaction(x, y, [&](auto& xprime, auto& yprime){
        modify xprime here (might throw)
        modify yprime here (might throw)
        …
    });
} catch (etc.) { … }
```

The assignment operator example would be rewritten as follows:

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

The `swap_assign` feature takes care of the boilerplate of exception-safe assignment and also handles the somewhat complicated allocator propagation traits.

Also proposed is a `get_allocator(x)` function template that returns the allocator for `x`, if it has one, and the default allocator otherwise. This primitive functionality is useful for implementing the other two templates, but is useful on its own and is thus described explicitly.

Target publication

These functions can be targeted for C++20 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

An earlier revision of this paper proposed two other function templates:

- `copy_swap_helper(x)` returned a copy of x using x's allocator even if the allocator would not normally propagate on copy construction.
- `copy_swap_helper(x, y)` returned a copy of x using y's allocator.

Both functions would work as normal copy constructors if `x` does not use an allocator.

The one-argument form of `copy_swap_helper` was removed because the `copy_swap_transaction` function expressed the idiom more cleanly.

The two-argument form of `copy_swap_helper` was removed because the only known use for such a function was for the copy-swap assignment idiom, and even then it did the wrong thing in the presence of some propagation traits. Thus, I encapsulated the entire idiom, including the correct use of propagation traits, into `swap_assign`, instead.

The functionality of both versions of `copy_swap_helper` can be implemented simply using the `make_using_alloctor` template proposed in P0591, combined with `get_allocator`, proposed here. This fact further reduces the motivation for `copy_swap_helper`.

Proposed Wording

This text is relative to the Library Fundamentals TS Version 2 DTS (LFTS 2), N4617.

Requests for guidance are highlighted yellow.

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 Transaction	TBD	`copy_swap_transaction`	201707	`<experimental/memory>`

Add to header `<experimental/memory>` synopsis:

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

```
template <class T>
  see-below get_allocator(T&& x);

template <class T, class... Rest>
  void copy_swap_transaction(T& t, Rest&&... rest);

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>
  see-below get_allocator(T&& x);
```

Returns: `x.get_allocator()` if that expression is well-formed; otherwise `allocator<byte>{}`.

Consistent with P0339, it might be better if the default return value were `pmr::polymorphic_allocator<byte>{}`. Thoughts?

It is probably reasonable to have `get_allocator()` be a customization point. What wording magic is needed for that?

```
template <class T, class... Args>
  void copy_swap_transaction(T& t, Args&&... args);
```

Requires: The `args` parameter pack shall have at least one element. All but the last element of `args` shall be lvalue references comprising a partial parameter pack `V&...v`. Each element of `v...` shall be *swappable* ([swappable.requirements] in C++17). The last element of `args` shall be an object `f` of type `F` such that `std::forward<F>(f)(v...)` is well-formed.

Effects: Let `v1, v2, …, vN` of types `V1&, V2&, …, VN&`, be the first N elements of parameter pack `args...`, where N is one less than `sizeof…(Args)`, and let value `f`, of type `F`, be the last argument in `args...`. For each i in 1…N, constructs `vi'` of type `Vi` by *uses-allocator construction* with allocator `get_allocator(vi)` and argument `vi` ([allocator.uses.construction] in C++17). Invokes `std::forward<F>(f)(v1', v2, …, vN')`. Then for i in N…1, invokes `swap(vi, vi')` in the context described by the swappable requirement.

Throws: nothing unless a constructor, swap, or invocation of `f` throws. [*Note:* Using arguments for which `swap` does not throw ensures that the values referenced by the first N arguments are modified only if `f` succeeds without throwing. – *end note*]

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

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

— If `get_allocator(lhs)` is well formed and `uses_allocator_v<T, decltype(get_allocator(lhs))>` 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 `get_allocator(rhs)` and argument `std::forward<T>(rhs)`. [*Note*: if the allocator's `propagate_on_container_swap` trait is false, then the `swap(lhs, R)` might produce unexpected results, including undefined behavior – *end note*]
- Otherwise R is an object of type T constructed by *uses-allocator construction* ([allocator.uses.construction] in the C++ standard) with allocator `get_allocator(lhs)` and argument `rhs`.

— Otherwise, R is `rhs`.

Returns: `lhs`

Remarks: The invocation of `swap` occurs in the context described for the swappable requirements ([swappable.requirements] in C++17).

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

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

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

- If `allocator_traits<decltype(get_allocator(lhs))>::propagate_on_container_move_assignment::value` is `true`, then R is `T(std::move(rhs))`. [*Note*: if the allocator's `propagate_on_container_swap` trait is false, then the `swap(lhs, R)` might produce unexpected results, including undefined behavior – *end note*]
- Otherwise R is an object of type T constructed by *uses-allocator construction* ([allocator.uses.construction] in the C++ standard) with allocator `get_allocator(lhs)` and argument `std::move(rhs)`.

— Otherwise, R is `rhs`.

Returns: `lhs`

Remarks: The invocation of `swap` occurs in the context described for the swappable requirements ([swappable.requirements] in C++17).