Document Number: P1456R0
Date: 2019-01-25
Audience: Library Evolution Working Group
Author: Casey Carter
Reply to: casey@carter.net

Move-only views

1 Scope

The `View` concept requires that its models are `Copyable`, which excludes some useful `Range` types - notably including coroutine generators. This proposal suggests relaxing the copyability requirement to allow `Range` types which are `Movable` but not `Copyable` to model `View` and hence be first-class citizens in view composition so as to be more usable with the ranges library.

Table 1 — Tony Table

Before	**After**

`return view::iota(0, 42)`
`  | view::filter(is_even)`
`  | view::transform([](int i) -> auto& {`
`      static auto r = int_range_generator(i);`
`      return r; // YOLO`
`    })`
`  | view::join;`
	`return view::iota(0, 42)`
	`  | view::filter(is_even)`
	`  | view::transform([](int i) {`
	`      return int_range_generator(i);`
	`    })`
	`  | view::join;`

`struct fib_view {`
`  struct iterator {`
`    using difference_type = int;`
`    using value_type = int;`
 	
`    int a = 0, b = 1;`
 	
`    int operator*() const { return a; }`
`    iterator& operator++() {`
`      a = std::exchange(b, a + b);`
`      return *this;`
`    }`
`    void operator++(int) { ++*this; }`
`  };`
 	
`  auto begin() const { return iterator{}; }`
`  auto end() const { return std::unreachable; }`
`};`
`// ...`
`return fib_view{} | view::take(20);`
	`generator<int> fib() {`
	`  int a = 0, b = 1;`
	`  while (true)`
	`    co_yield std::exchange(a, std::exchange(b, a + b));`
	`}`
	`// ...`
	`return fib() | view::take(20);`

1.1 Revision History

1.1.1 Revision 0

— Inserted many words into an empty document.

2 Problem description

2.1 background

P0896R4 “The One Ranges Proposal” [2] introduced the `View` concept into the C++working draft. `View` refines `Semiregular` which refines `Copyable` in order that `View`s may be used in ways similar to `Iterator`s which also refine `Semiregular`. `Semiregular`ity combined with the constant-time complexity requirement on `View` and `Iterator` operations allows them to copied freely by range adaptors and the range adaptor closure object machinery used in `View` composition. Since general `Range` types aren’t required to support such usage, they are in some sense second-class citizens in `View` composition.

C++11 added move-only types to the language. Parts of the Standard Library have adapted to better support move-only types, for example, we can store them in containers. Other parts of the library have been reluctant to do so: the algorithms - with the exception of the serial overload of `std::for_each` - still require copyable function objects, for example, and we still don’t have a mechanism to type erase move-only function objects. It behooves us to consider whether the Ranges view composition machinery added by P0896R4 should support move-only types by admitting move-only views.

Some examples to consider:

— `single_view` ([range.single.view]) requires its element to model `CopyConstructible`. The element is stored inside an `optional`-like wrapper in the `single_view` object, so the element must model `CopyConstructible` in order for the `single_view` to model `Semiregular`. If `View`’s copyability requirement were relaxed, `single_view` could support types that only model `MoveConstructible`.

— `filter_view` and `transform_view` each hold a function object. Those function objects are similarly required to be `CopyConstructible` only so the view can model `Semiregular`.

2.2 Why views but not function object arguments?

We’ve managed to put off supporting move-only types in standard library algorithms on the basis that there’s a simple workaround: add a layer of indirection! If you can’t call `std::ranges::find_if(myvec, move_only_predicate)` with your move-only function object, call `std::ranges::find_if(myvec, std::ref(move_only_predicate))` instead. This solution typically isn’t a burden since calls to algorithms typically occur in straight-line code that already has named objects for the range / iterator arguments. If you want to pass an rvalue move-only function object to an algorithm, stuff it into a named object as well and use `std::ref`.

View composition differs fundamentally from a series of algorithm calls. Instead of a series of statements interleaved with declarations, a view composition expression is a composition of nested function calls, possibly disguised with the `|` operator to give it a more linear appearance, that yields a view. There’s nowhere to store intermediate results and no mechanism to name temporaries in "the middle" of such an expression. It’s necessary to find a place to store the move-only range, pass an lvalue into view composition, and somehow ensure that the stored range outlives the composed view. This can be challenging when a function wants to return the result of composition of a local move-only range with some sequence of range adaptors.

Another important difference is that `View` is a concept that’s exported for users to utilize in their own programs, whereas the requirement that the algorithms need copyable function objects is only words in a specification. WG21 can relax a requirement expressed in specification text any time we like simply by changing the words - conforming implementations must ensure they handle the relaxed requirements, but there’s no possiblity of breaking user programs. Conversely, the set of requirements expressed by a concept in the Standard can never change without potential breakage: if we strengthen a concept, existing callers of standard library components constrained with that concept may be broken. If we weaken a concept, third-party library components constrained with that concept are broken by suddenly becoming underconstrained. We have one chance to get concepts right.

2.3 Coroutines on the horizon

Generators are the intersection of coroutines and ranges. A generator is, simply enough, a coroutine that models the `Range` concept. A user writes a function whose return type is a generator type that `co_yields` successive elements of the range (LIVE):

```
generator<int> fib() {
  int a = 0, b = 1;
  while (true)
    co_yield std::exchange(a, std::exchange(b, a + b));
}
```

which is substantially more concise than the equivalent direct implementation of such an input range (LIVE):

```
struct fib_view {
  struct iterator {
    using difference_type = int;
    using value_type = int;

    int a = 0, b = 1;

    int operator*() const { return a; }
    iterator& operator++() {
      a = std::exchange(b, a + b);
      return *this;
    }
    void operator++(int) { ++*this; }
  };

  auto begin() const { return iterator{}; }
  auto end() const { return std::unreachable; }
};
```

Note that the direct range implementation models `View`, so it can easily pariticipate in view composition - but the same is not true for the generator implementation.

Coroutine frames are not copyable resources, so it's not possible to implement a generator that can produce truly independent copies. It is possible to implement a generator that is as `Semiregular` as are input iterators[1], but only by invoking the same spooky-action-at-distance that input iterators have in which operations performed on an input iterator (or generator) can potentially invalidate the copies. `Movable`-but-not-`Copyable` input iterators (or generators) would avoid this issue. While it may be too late for us to solve the problem for input iterators, it would seem a shame to repeat the design error with generators.

2.4 What about range adaptors that want to copy `View`s?

The range adaptors in the working draft - and the range adaptors in range-v3 - only copy views in one circumstance: when a user calls the `base()` member function to retrieve a copy of the underlying view being adapted. None of the adaptors copy views in the process of their normal operation. In fact, no one has suggested to the author a potential design for a range adaptor which *does* need to copy a view.

On the other hand, we *do* have concrete examples of types that would like to model `View` which are not copyable. The evidence suggests that we should relax the copyability requirement of `View` now, and if and when an adaptor is discovered which does need to copy a view it can be constrained with `View<T> && Copyable<T>` along with the added semantic requirement that copies are O(1). We could introduce a `CopyableView` concept with these requirements when that day comes to ease using this constraint, although the lack of any suggestion of an adaptor that needs the constraint suggests it would be premature to standardize such a concept now.

2.5 Does this mean containers can be views?

Potentially yes, although there are some messy details. `View` should require O(1) destruction as well as the current requirement for O(1) copies and moves - which lack the proposed wording below corrects - which rules out many containers. Allocator-aware containers also notably only have O(1) move assignment when their allocator has specific properties. This paper does not propose any changes to allow `View` to accept container types that would otherwise be rejected by the `enable_view` heuristic, but that is a potential avenue for future work.

1) With thanks to Lewis Baker.

Eric Niebler has expressed concerns to the author in private communication about such a change muddying the Ranges design, with which the author agrees: we need to tread very carefully here.

3 Proposal

Put simply, the proposal is to remove the copyability requirement from the `View` concept as defined in [range.view]:

```
template<class T>
  concept View =
    Range<T> && Semiregular<T> && enable_view<T>;
```

by decomposing `Semiregular<T>` into `Copyable<T> && DefaultConstructible<T>` and then replacing `Copyable` with `Movable`:

```
template<class T>
  concept View =
    Range<T> && Movable<T> && DefaultConstructible<T> && enable_view<T>;
```

This allows for move-only types to model `View` without excluding any types which model the prior formulation of the concept.

Note that a similar relaxation that only applies to single-pass (input and output) views has been implemented in range-v3 [1] since June of 2017 along with the experimental range generator implementation.

To redress the problem that the `base()` members of the range adaptors return copies of the underlying view, we propose that each such `base()` member be replaced to by two overloads: a `const`-qualified overload that requires the type of the underlying view to model `CopyConstructible`, and a `&&`-qualified overload that extracts the underlying view from the adaptor (thus leaving it invalid).

4 Technical specifications

Change [range.req.general]/2 as follows [*Note:* There's a drive-by edit here to require O(1) destruction: the intent has always been that `View`s have only constant-time operations, we apparently forgot that destruction is an operation. — *end note*] :

> ... The `View` concept specifies requirements on a `Range` type with constant-time ~~copy and assign~~move operations and destruction.

Change [range.view] as follows:

1 The `View` concept specifies the requirements of a `Range` type that has constant time ~~copy,~~ move construction, move assignment, and destruction ~~operators~~; that is, the cost of these operations is not proportional to the number of elements in the `View`.

2 [*Example:* Examples of `View`s are:

(2.1) — A `Range` type that wraps a pair of iterators.

(2.2) — A `Range` type that holds its elements by `shared_ptr` and shares ownership with all its copies.

(2.3) — A `Range` type that generates its elements on demand.

Most containers ([containers]) are not views since ~~copying~~destruction of the container ~~copies~~destroys the elements, which cannot be done in constant time. — *end example*]

```
template<class T>
  inline constexpr bool enable_view = see below;

template<class T>
  concept View =
    Range<T> && Semiregular<T> && enable_view<T>;

  template<class T>
    concept View =
      Range<T> && ~~Semiregular<T>~~Movable<T> && DefaultConstructible<T> && enable_view<T>;
```

Change [range.filter.view] as follows:

```
namespace std::ranges {
  template<InputRange V, IndirectUnaryPredicate<iterator_t<V>> Pred>
    requires View<V> && is_object_v<Pred>
  class filter_view : public view_interface<filter_view<V, Pred>> {
    [...]
    constexpr filter_view(R&& r, Pred pred);

    constexpr V base() const requires CopyConstructible<V> { return base_; }
    constexpr V base() && { return std::move(base_); }

    constexpr iterator begin();
    [...]
```

[...]

~~`constexpr V base() const;`~~

3 ~~*Effects:* Equivalent to: `return base_;`~~

Change [range.transform.view], [range.take.view], [range.common.view], and [range.reverse.view] similarly.

Change [range.join.view] as follows [*Note:* This is a drive-by fix to add `base` which was unintentionally omitted from `join_view` and `split_view` in P0896R4. — *end note*] :

```
namespace std::ranges {
  template<InputRange V>
    requires View<V> && InputRange<iter_reference_t<iterator_t<V>>> &&
             (is_reference_v<iter_reference_t<iterator_t<V>>> ||
              View<iter_value_t<iterator_t<V>>>)
  class join_view : public view_interface<join_view<V>> {
    [...]
    template<InputRange R>
      requires ViewableRange<R> && Constructible<V, all_view<R>>
    constexpr explicit join_view(R&& r);

    constexpr V base() const requires CopyConstructible<V> { return base_; }
    constexpr V base() && { return std::move(base_); }

    constexpr auto begin() {
    [...]
```

[...]

Add the same definitions to [range.split.view] before the first definition of `begin`.

Change [range.join.iterator]/6 as follows [*Note:* Yes, I claimed above that “none of the adaptors copy a view in their normal operation” despite that this is an occurrence of exactly that which should properly be a move. Sue me. — *end note*] [*Note:* This includes a drive-by simplification of a return type from `decltype(auto)` to `auto&` to clarify that `update_inner` always returns an lvalue. — *end note*] :

```
auto update_inner = [this](iter_reference_t<iterator_t<Base>> x) -> ~~decltype(auto)~~auto& {
  if constexpr (ref_is_glvalue) // x is a reference
    return (x);                 // (x) is an lvalue
  else
    return (parent_->inner_ = view::all(std::move(x)));
};

for (; outer_ != ranges::end(parent_->base_); ++outer_) {
  auto& inner = update_inner(*outer_);
  [...]
```

Bibliography

[1] Eric Niebler. Range-v3. https://github.com/ericniebler/range-v3. Accessed: 2019-1-18.

[2] Eric Niebler, Casey Carter, and Christopher Di Bella. P0896R4: The one ranges proposal, 11 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0896r4.pdf.