Constraining Readable Types

Document #: P1878R1
Date: 2019-11-06
Project: Programming Language C++
LWG
Reply-to: Eric Niebler
<eniebler@fb.com>
Casey Carter
<cacarter@microsoft.com>

1 Abstract

There are a number of serious issues in the current definitions of the `readable` and `indirectly_swappable` concepts and their associated types that prevent them from properly constraining the algorithms with which they are constrained. Several of these are the subject of US NB comments. They all have to do with whether or not `const`- and/or reference-qualified types (lvalues and rvalue) model `readable` and/or `indirectly_swappable`.

This paper treats all these problems together and suggests a simple fix for the problems, addressing the NB concerns.

2 The Problems

2.1 Problem #1: The `readable` concept is sensitive to `const`-ness and value category

The `readable` concept is what gives input iterators their "read-ability" via the unary `operator*` syntax, and the `iter_` associated types (`iter_value_t`, `iter_reference_t`, and `iter_rvalue_reference_t`). The formulation in [N5410] is as follows:

```
template<class In>
  concept readable =
    requires {
      typename iter_value_t<In>;
      typename iter_reference_t<In>;
      typename iter_rvalue_reference_t<In>;
    } &&
    common_reference_with<iter_reference_t<In>&&, iter_value_t<In>&> &&
    common_reference_with<iter_reference_t<In>&&, iter_rvalue_reference_t<In>&&> &&
    common_reference_with<iter_rvalue_reference_t<In>&&, const iter_value_t<In>&>;
```

This is testing that the associated types are well-formed. The presence of a unary `operator*` is implied by the well-formed-ness of the `iter_reference_t` associated type, which is specified as:

```
template<dereferenceable T>
  using iter_reference_t = decltype(*declval<T&>());
```

There are a couple of problems with this formulation of `readable<In>`:

1. Only an lvalue of type `In` is required to be readable. Rvalues are not required to be readable, nor are `const` lvalues. The STL as specified assumes that the value category and `const`-ness of an iterator does not affect whether it can be dereferenced. The `readable` concept should capture this.
2. Nothing about this formulation is requiring that, say, `iter_value_t<X>` names the same type as `iter_value_t<const X>`; or that `iter_reference_t<X>` names the same type as `iter_referece_t<const X>`. Most code that is generic over input iterators is not equipped to deal with iterator types that violate these assumptions. The `readable` concept should require that the associated types are insensitive to top-level `const` and reference qualification.

The fix would be for the `readable` concept to add additional requirements to test all the permutations of `const` and reference qualification for all the associated types and also for the unary `operator*` expression. That is obviously prohibitively expensive, both in specification complexity as well as with compile-time resources.

This specification difficulty is not new, and the concepts in the Standard Library make use of a syntactic convention to opt-in to all such permutations of `const` qualification and value category: *implicit expression variations*.

The exact meaning of implicit expression variations, and how a concept definition opts-in to these extra syntactic and semantic constraints, are specified in [concepts.equality]/p6-8. In short, if the local parameters declared in a *requires-expression* are `const`-qualified, any use of that parameter as the operand of a required expression generates implicit variations of that required expression that are *also* required, each of which uses the operand with a different `const` qualification and/or value category. Implementations are not required to syntactically enforce these extra requirements, but a type only models the concept if it supports all the required expressions, explicit and implicit.

The fix to the `readable` concept is to reformulate it so that we get the benefit of implicit expression variations. See below:

```
template<class In>
  concept readable-impl = // exposition-only
    requires(const In in) {
      typename iter_value_t<In>;
      typename iter_reference_t<In>;
      typename iter_rvalue_reference_t<In>;
      { *in } -> same_as<iter_reference_t<In>>;
      { iter_move(in) } -> same_as<iter_rvalue_reference_t<In>>;
    } &&
    common_reference_with<iter_reference_t<In>&&, iter_value_t<In>&> &&
    common_reference_with<iter_reference_t<In>&&, iter_rvalue_reference_t<In>&&> &&
    common_reference_with<iter_rvalue_reference_t<In>&&, const iter_value_t<In>&>;

template<class In>
  concept readable =
    readable-impl<remove_cvref_t<In>>;
```

The above change uses the implicit variations to enforce that all permutations of `const` qualification and value category on an iterator give the same semantics and have the same `iter_reference_t` and `iter_rvalue_reference_t`. However, it says nothing about `iter_value_t` since that type is not a part of any of the required expressions that involve the local parameter `in` declared in the *required expression*.

We can fix this by redefining the `iter_value_t` alias to strip top-level `const` and reference qualification before using it to instantiate `std::iterator_traits` and `std::readable_traits`. See the section Proposed Resolution.

After this change, a type such as `std::optional` no longer models `readable` because the return type of its unary `operator*` member is different depending on the `const`-ness of the `std::optional`.

Presently, we have no examples of algorithms that are generic over pointer-and-iterator-like things and optional-like things, so the lack of a concept that can be used to constrain such algorithms is not troubling. However, to indicate that `readable` is not usable for constraining operations on types that do not represent an indirection, we might consider renaming `readable` to `indirectly_readable`. This paper does not propose that, but it would probably be worth polling.

2.2 Problem #2: `indirectly_swappable` only tests that `iter_swap` is callable with lvalue iterators

The formulation of `indirectly_swappable` in [N5410] is as follows:

```
template<class I1, class I2 = I1>
  concept indirectly_swappable =
    readable<I1> && readable<I2> &&
    requires(I1& i1, I2& i2) {
      ranges::iter_swap(i1, i1);
      ranges::iter_swap(i2, i2);
      ranges::iter_swap(i1, i2);
      ranges::iter_swap(i2, i1);
    };
```

This requires that lvalue expressions of the two iterator types are indirectly swappable by passing the lvalues to the `iter_swap` customization point. However, this concept does not require that *rvalues* can be passed to `iter_swap`. Permuting algorithms are expected to frequently be implemented in terms of expressions such as `iter_swap(i+n, j+m)`, making this oversight somewhat embarrassing. We do not intend to over-constrain the algorithms by requiring authors to assign iterator expressions to local variables in order to swap the elements they denote.

An earlier formulation of `indirectly_swappable` (in [P0022R2]) made use of *implicit expression variations* ([concepts.equality]/p6) to handle all the necessary combinations of `const`- and non-`const`-qualification and lvalue and rvalue categories for the `iter_swap` arguments, as shown below (in the syntax of the Concepts TS):

```
template <class I1, class I2>
concept bool IndirectlySwappable() {
  return Readable<I1>() && Readable<I2>() &&
    requires(const I1 i1, const I2 i2) { [ Editor's note: Implicit expression variations here. ]
      iter_swap(i1, i2);
      iter_swap(i2, i1);
      iter_swap(i1, i1);
      iter_swap(i2, i2);
    };
}
```

It was changed in a misguided effort to generalize `iter_swap` to make it possible to swap instances of types that use pointer-like syntax but do not represent an indirection, like `std::optional`.

The fix is to restore the formulation that made use of implicit expression variations. `iter_swap` will still be usable with types like `std::optional`, but an algorithm making use of such syntax cannot be constrained with `indirectly_swappable` after the proposed change.

2.3 Problem #3: `shared_ptr<int>&` does not satisfy `readable`

In the formulation of `readable` in [N5410], the type `std::shared_ptr<int>` models `readable`, but `std::shared_ptr<int>&` does not. This is an unintended consequence of the particular implementation of `iter_value_t` and how it dispatches to `std::readable_traits`. In particular, in [N5410] `readable` is defined in terms of `iter_value_t<In>` which does not strip top-level reference qualifiers from `readable_traits<In>` before looking for a nested `::value_type`.

The specification of `readable_traits<In>` gives it a nested `::value_type` if `In::value_type` is well-formed and names a type. That is not the case if `In` is a reference type.

One possible fix is to define `readable<In>` in terms of `iter_value_t<remove_reference_t<In>>`. However, the fix to `readable` described above in Problem #1 – that is, changing the definition of `iter_value_t` to strip top-level cv and ref qualification – suffices to fix this problem as well.

3 Implementation Experience

The proposed resolution below has been applied to range-v3. All tests passed after the change.

After making the suggested change to the `readable` concept, someone filed a bug about `std::optional` no longer satisfying `readable`. The code demonstrating the problem is shown below:

```
// This compiled before the change but not after.
views::generate( [](){ return std::optional<int>{ 1 }; } ) | views::indirect;
```

In range-v3 `views::indirect` is a view that transforms a range of `readable` values into a range of the values to which the `readable`s refer. Implicit in `views::indirect` is the assumption that the `readable`s actually represent an indirection; that is, `views::indirect` assumes the reference returned by a `readable` object's `operator*` is valid even after the `readable` object itself is destroyed. This is certainly *not* the case for `std::optional` or any other readable-like types that do not represent an indirection. In short, this change helped the user to find a source of undefined behavior in their code, which reinforces the authors' belief that this change is correct.

4 Proposed Resolution

The following proposed resolution resolves the three issues described above in a consistent way.

[Editor's note: Change [iterator.synopsis] as follows:]

```
...
template<class T>
  using iter_difference_t = see below ;

// 23.3.2.2,indirectly_readable traits
template<class> struct indirectly_readable_traits;
template<class T>
  using iter_value_t = see below;

...

// 23.3.4.2, conceptindirectly_readable
template<class In>
  concept indirectly_readable = see below;

template<readable T>
  using iter_common_reference_t =
    common_reference_t<iter_reference_t<T>, iter_value_t<T>&>;

// 23.3.4.3, conceptindirectly_writable
template<class Out, class T>
  concept indirectly_writable = see below;
```

[Editor's note: Change [incrementable.traits]/p2 as follows (to be consistent with the change to `iter_value_t` below):]

2 The type `iter_difference_t<I>` denotes

2.1 `incrementable_traits<remove_cvref_t<I>>::difference_type` if `iterator_traits<remove_cvref_t<I>>` names a specialization generated from the primary template, and

2.2 `iterator_traits<remove_cvref_t<I>>::difference_type` otherwise.

[Editor's note: Change the stable name [readable.traits] to [indirectly.readable.traits], and change the section as follows:]

1 To implement algorithms only in terms of indirectly readable types, it is often necessary to determine the value type that corresponds to a particular indirectly readable type. Accordingly, it is required that if `R` is the name of a type that models the `indirectly_readable` concept (23.3.4.2), the type

```
iter_value_t<R>
```

be defined as the indirectly readable type's value type.

```
template<class> struct cond-value-type { }; // exposition only
template<class T>
  requires is_object_v<T>
struct cond-value-type {
  using value_type = remove_cv_t<T>;
};

template<class> struct indirectly_readable_traits { };

template<class T>
struct indirectly_readable_traits<T*>
  : cond-value-type<T> { };

template<class I>
  requires is_array_v<I>
struct indirectly_readable_traits<I> {
  using value_type = remove_cv_t<remove_extent_t<I>>;
};

template<class I>
struct indirectly_readable_traits<const I>
  : readable_traits<I> { };

template<class T>
  requires requires { typename T::value_type; }
struct indirectly_readable_traits<T>
  : cond-value-type<typename T::value_type> { };

template<class T>
  requires requires { typename T::element_type; }
struct indirectly_readable_traits<T>
  : cond-value-type<typename T::element_type> { };

template<class T> using iter_value_t = see below;
```

2 The type `iter_value_t<I>` denotes

2.1 `indirectly_readable_traits<remove_cvref_t<I>>::value_type` if `iterator_traits<remove_cvref_t<I>>` names a specialization generated from the primary template, and

2.2 `iterator_traits<remove_cvref_t<I>>::value_type` otherwise.

3 Class template `indirectly_readable_traits` may be specialized on program-defined types.

4 [*Note:* Some legacy output iterators define a nested type named `value_type` that is an alias for `void`. These types are not `indirectly_readable` and have no associated value types. — *end note*]

5 [*Note:* Smart pointers like `shared_ptr<int>` are `indirectly_readable` and have an associated value type, but a smart pointer like `shared_ptr<void>` is not `indirectly_readable` and has no associated value type. — *end note*]

[Editor's note: Globally replace "`readable_traits`" with "`indirectly_readable_traits`"]

[Editor's note: The following change to [iterator.cust.swap]/p2 is a clean-up made possible by the above change to `iter_value_t`:]

2 Let *`iter-exchange-move`* be the exposition-only function:

```
template<class X, class Y>
  constexpr iter_value_t<remove_reference_t<X>> iter-exchange-move(X&& x, Y&& y)
    noexcept(noexcept(iter_value_t<remove_reference_t<X>>(iter_move(x))) &&
      noexcept(*x = iter_move(y)));
```

3 *Effects:* Equivalent to:

```
iter_value_t<remove_reference_t<X>> old_value(iter_move(x));
*x = iter_move(y);
return old_value;
```

[Editor's note: Change the stable name of [iterator.concept.readable] to [iterator.concept.indirectly.readable], and change the the section as follows:]

1 Types that are indirectly readable by applying `operator*` model the `indirectly_readable` concept, including pointers, smart pointers, and iterators.

```
template<class In>
  concept readableindirectly-readable-impl =
    requires(const In in) {
      typename iter_value_t<In>;
      typename iter_reference_t<In>;
      typename iter_rvalue_reference_t<In>;
      { *in } -> same_as<iter_reference_t<In>>;
      { iter_move(in) } -> same_as<iter_rvalue_reference_t<In>>;
    } &&
    common_reference_with<iter_reference_t<In>&&, iter_value_t<In>&> &&
    common_reference_with<iter_reference_t<In>&&, iter_rvalue_reference_t<In>&&> &&
    common_reference_with<iter_rvalue_reference_t<In>&&, const iter_value_t<In>&>;

template<class In>
  concept indirectly_readable =
    indirectly-readable-impl<remove_cvref_t<In>>;
```

2 Given a value `i` of type `I`, `I` models `indirectly_readable` only if the expression `*i` is equality-preserving. [*Note:* The expression `*i` is indirectly required to be valid via the exposition-only *dereferenceable* concept (23.2). — *end note*]

[Editor's note: Globally replace "`readable`" (the concept) with "`indirectly_readable`"]

[Editor's note: Change stable name [iterator.concept.writable] to [iterator.concept.indirectly.writable] and globally replace all occurrences of "`writable`" (the concept) with "`indirectly_writable`".]

[Editor's note: Change [alg.req.ind.swap] as follows:]

1 The `indirectly_swappable` concept specifies a swappable relationship between the values referenced by two indirectly readable types.

```
template<class I1, class I2 = I1>
  concept indirectly_swappable =
    indirectly_readable<I1> && indirectly_readable<I2> &&
    requires(~~I1&~~const I1 i1, ~~I2&~~const I2 i2) {
      ranges::iter_swap(i1, i1);
      ranges::iter_swap(i2, i2);
      ranges::iter_swap(i1, i2);
      ranges::iter_swap(i2, i1);
    };
```

[Editor's note: Change [algorithms.requirements]/p12 as follows (this is merely a simplification):]

12 In the description of the algorithms, operator `+` is used for some of the iterator categories for which it does not have to be defined. In these cases the semantics of `a + n` are the same as those of

```
auto tmp = a;
for (; n < 0; ++n) --tmp;
for (; n > 0; --n) ++tmp;
return tmp;
```

Similarly, operator `-` is used for some combinations of iterators and sentinel types for which it does not have to be defined. If `[a, b)` denotes a range, the semantics of `b - a` in these cases are the same as those of

```
iter_difference_t<~~remove_reference_t<~~decltype(a)~~>~~> n = 0;
for (auto tmp = a; tmp != b; ++tmp) ++n;
return n;
```

and if `[b, a)` denotes a range, the same as those of

```
iter_difference_t<~~remove_reference_t<~~decltype(b)~~>~~> n = 0;
for (auto tmp = b; tmp != a; ++tmp) --n;
return n;
```