Iterator Difference Type and Integer Overflow

Document #: D1775R0
Date: 2019-04-17
Project: Programming Language C++
Library Evolution Working Group
Library Working Group
Reply-to: Eric Niebler
<eric.niebler@gmail.com>

Contents

1 Introduction

This paper seeks to redress a long-standing shortcoming of iterators with respect to the potential for signed integer overflow in an iterator's difference type. It does so in a low-impact way for C++20 by permitting user-defined (but not program-defined) integer-like types to be used as an iterator's difference type. This leeway is used to solve a smattering of serious bugs in C++20's `iota_view`, and can be used in the future to avoid undefined behavior when we inevitably add infinite ranges to the Standard Library.

2 Motivation and Scope

The `WeaklyIncrementable` concept in the current Working Draft requires all incrementable types (including iterators) to declare a `difference_type`. The difference type is required to be a "signed integer type" ([iterator.requirements.general]/p1). This obviously presents a problem for infinite, near-infinite, or possibly-infinite ranges, which can potentially overflow their difference types, leading to undefined behavior.

This problem immediately manifests in C++20's `iota_view`, which builds a range out of either a `WeaklyIncrementable` value (an unbounded range starting at that value), or a pair of `Incrementable` values (denoting a half-open range). If the incrementable type in question is the highest-precision integer (signed or unsigned) supported by the platform, it is impossible to pick a signed integer type to use as the `iota_view`'s difference type such that that it is valid over the range's entire domain. Merely subtracting the beginning of such a range from the end can invoke undefined behavior.

```
iota_view v {size_t(0), SIZE_MAX};
auto d = v.end() - v.begin(); // Undefined behavior
```

Due to the requirement that difference type be a (built-in) signed integral type, there is no recourse.

Worse, the dual requirements that (a) all iterators have a difference type, and (b) the difference type is a signed integral type are baked into the iterator concept hierarchy which, once C++20 is final, is fixed and immutable for all time.

2.1 Design Principles

(As stolen from [Sutter] P0707:)

The primary design goal is conceptual integrity [Brooks 1975], which means that the design is coherent and reliably does what the user expects it to do. Conceptual integrity's major supporting principles are:

— **Be consistent:** Don't make similar things different, including in spelling, behavior, or capability. Don't make different things appear similar when they have different behavior or capability.

— **Be orthogonal:** Avoid arbitrary coupling. Let features be used freely in combination.

— **Be general:** Don't restrict what is inherent. Don't arbitrarily restrict a complete set of uses. Avoid special cases and partial features.

In addition to the above general principles, given the stability requirements on the Iterators and Ranges design at this late stage of C++20, and the long-term importance of getting this right, for this paper, I add additional design principles:

— **Be non-disruptive:** This design seeks to keep the current Iterator and Ranges design intact to the extent possible to minimize the risk of unintended consequences.

— **Be extensible:** We consider an incomplete solution acceptable provided we have reason to believe a full solution can be added later in an ABI-conpatible way.

2.2 Proposed Solution

The solution is to modify the `WeaklyIncrementable` concept to permit integer-like user-defined types to be used as an iterator's difference type. We do this in such a way that (a) only implementors of the Standard Library can define and use such types for now, and (b) is forward ABI compatible should we decide at a later time to extend this to program-defined integer-like types. ("User-defined" and "program-defined" are terms used in the standard to denote class/union/enum types and class/union/enum types that are not part of the Standard Library implementation, respectively.)

This extra dispensation given to Standard Library implementors can be immediately used to eliminate the potential for integer overflow from the `iota_view`. For every built-in integer type `N`, the `iota_view`'s difference type would be the signed integer type of the next-highest width supported by the platform (or the type of `N{0} - N{0}` if that has enough bits). When there is no such greater-precision built-in integral type, the Standard Library would be permitted to use some unspecified type that is sufficiently integer-like and has enough width to represent the full range.

Naturally we have to specify what we mean by "sufficiently integer-like." Rather than specify the requirements of integer-like types as a standard concept, where it would be fixed and immutable for all time, we specify it as an exposition-only trait, *`is-integer-like<T>`*, that is `true` for the built-in integral types and some set of implementation-defined types – provided they exhibit sufficiently integer-like behaviors. Those syntactic and semantic requirements are described in prose, which is a more forgiving specification medium than concepts.

2.2.1 Anticipated Problems

There is an obvious infinite regress problem in the proposed solution. Any user-defined integer type should itself be incrementable, and thus should have a difference type that can represent the full range, which would require one extra bit. And that type would also need a higher-precision difference type, and so on, up to an infinite precision integer type. We do not want to commit implementors to writing an infinite precision integer type (although that would be cool), so we cut off the infinite regress by only giving built-in integers a difference type by default; hence, any user-defined integer types fail to satisfy `WeaklyIncrementable`. `iota_view<range_difference_t<iota_view<uintmax_t>>` would simply fail to compile.

2.2.2 Additional Use Cases

Over the life of the [range-v3] project, several people have asked for the library to support program-defined integer types, as shown by the following feature requests:

— "Integral concept overly simple" <https://github.com/ericniebler/range-v3/issues/356>
— "view/iota over opaque integers" <https://github.com/ericniebler/range-v3/issues/514>
— "Relax difference_type to support user-defined wrappers over `Integer` types" <https://github.com/ericniebler/range-v3/issues/591>

The resolution proposed in this paper would open the door for such extensions post-C++20.

Here are some of the things people have wanted to do with support for program-defined integer types:

1. Use a "big" integer type with more width than their platform provides.
2. Use a safe integer wrapper that debug-asserts on overflow.
3. Use a saturating integer wrapper that locks to "infinity" on overflow.
4. Use a wrapper around an unsigned integer that disables modular arithmetic (to be used as the return type of a `.size()` member, possibly).
5. Use a wrapper around a signed integer to assert if it is ever negative.

Of these, the first three are germane to iterators' difference types.

3 Implementation Experience

The proposed resolution has been fully implemented in the range-v3 library in the master branch. range-v3 makes heavy internal use of a façade template for defining iterator types. After defining the trait and modifying the concept as described, the façade was modified to give every iterator in range-v3 a program-defined integer-like difference type. This exposed the places in range-v3 where the code made assumptions about the iterator difference types. These problems were relatively few and easy to fix. The most common faulty assumption was that all difference types are inter-convertible.

In all, it required only a few days of work to get all of range-v3's tests passing with program-defined difference types.

4 Impact on the Standard

I consider the potential impact of the proposed solution on users and on implementors.

4.1 Impact on Users

The *immediate* impact on users is negligible: we change the difference type of no iterators currently in the field.

For users adopting C++20, the majority of iterators with which users will interact will use built-in types for their difference types. In the author's experience, most iterator-based code doesn't mention the iterator's difference type, the exception of course being generic algorithms that count things or that use divide-and-conquer strategies.

However, we expect `iota_view<size_t [, size_t]>` to be a quite common specialization, it being useful to enumerate other ranges. On platforms that do not have extended integral types larger than `size_t`, the difference type of that range's iterator will possibly be user-defined. (This proposal grants implementors the choice of whether or not to use a user-defined type in this case.)

In those rare cases in which a user encounters a user-defined integer type, there is a fair chance that they will need to make accommodation for such in their code. Here are the sorts of problems that the author encountered while hardening the range-v3 library against such types:

— **No implicit conversions:** All the built-in integral types are implicitly convertible to all the others. This is quite unsafe, and with a user-defined integral type we would hope to do better. The downside of using a safer integer-like type is that code that is sloppy about conversions would fail to compile.

— **Not required to be `WeaklyIncrementable`:** There is no requirement that a user-defined integer-like type specify what *its* difference type is. (If we were to require that, then by induction implementors would be on the hook to provide an infinite-precision integer-like type.) Therefore, `iota_view<iter_difference_t<I>>` for an arbitrary iterator `I` may in fact be ill-formed.

— **Cannot be passed to integer operations:** Most of the functions in `<cmath>`, for instance, are unlikely to be callable with such a type.

But probably the biggest issue is that an iterator that had a user-defined difference type would fail to satisfy the C++98 iterator requirements. Passing such an iterator to an algorithm in `std::`, for instance, is unlikely to succeed until implementors make explicit accommodations. (Experience shows this to be not difficult.)

4.2 Impact to Implementors

If this proposed resolution is accepted as specified, then there is no requirement that an implementor define or use a user-defined difference type anywhere in the Standard Library. And since the *`is-integer-like`* trait used to opt in is exposition-only, no program-defined iterator types can ever specialize this trait to return `true`. Therefore, an implementor can skirt the issue entirely.

Choosing this option, however, has implications. As stated in the problem description, simply doing `ranges::distance(iota_view{size_t(0), SIZE_MAX})` would result in undefined behavior. If the implementor chooses to address this problem with a user-defined integer-like type, their road is somewhat longer.

The largest impact in that case is increased testing burden. Any part of the Standard Library that makes direct or indirect use of an iterator's difference type would need to be tested with an iterator that has a user-defined difference type. This is not a trivial undertaking.

Those tests are likely to expose places in the Standard Library where the implementation is assuming the difference type is a built-in integral. For instance, it may assume that the difference type is implicitly convertible to `ptrdiff_t`. These are generally not hard to fix.

In all, it took roughly 2 days of work to modify the range-v3 library to support UDT integer types everywhere. I would expect the effort for Standard Library implementors to be commensurate.

5 Proposed Wording

[Editor's note: Add the following to [iterator.synopsis]:]

```
#include <concepts>

namespace std {
+  template<class T>
+    inline constexpr bool is-integer-like // exposition only
+      = see below ;
+  template<class T>
+    inline constexpr bool is-signed-integer-like // exposition only
+      = see below ;
   template<class T> using with-reference = T&; // exposition only
   template<class T> concept can-reference // exposition only
     = requires { typename with-reference<T>; };
   template<class T> concept dereferenceable // exposition only
     = requires(T& t) {
       { *t } -> can-reference; // not required to be equality-preserving
     };
...
```

[Editor's note: Following the synopsis add a new paragraph as follows:]

1 *is-integer-like*<T> is `true` if and only if `T` is an integer-like type (REF{iterator.concepts.general}). *is-signed-integer-like*<T> is `true` if and only if `T` is a signed integer-like type.

[Editor's note: Change [iterator.requirements.general]/p1 as follows:]

1 [...] For every iterator type `X`, there is a corresponding signed integer-like (REF) type called the *difference type* of the iterator.

[Editor's note: In [iterator.concepts.general], add a new paragraph p2 as follows]

2 A type `T` is *integer-like* if it models `Integral<T>` or if it is in a set of implementation-defined types that behave as integer types do, as defined below.

3 The *range* of an integer-like type is the continuous set of values over which it is defined. The values `0` and `1` are part of the range of every integer-like type. If any negative numbers are part of the range, the integer-like type is *signed*; otherwise, it is *unsigned*.

4 For every integer-like type `T`, let there be a hypothetical built-in integral type `U` of the same signedness with the smallest width ([basic.fundamental]/p1) capable of representing the same range of values. The type `U` has no integral promotions. The *width* of `T` is equal to the width of `U`.

5 Let `a` and `b` be objects of integer-like type `T`, let `x` and `y` be objects of type `U` as described above that represent the same values as `a` and `b` respectively, and let `c` be an lvalue of any built-in integral type.

(5.1) — For every unary operator `@` for which the expression `@x` is well-formed, `@a` shall also be well-formed and have the same value, effects, and value category as `@x` provided that value is within the range of `T`. The expression `@a` shall be constant time. If `@x` has type `bool`, so too does `@a`; if `@x` has type `U`, then `@a` has type `T`.

(5.2) — For every assignment operator `@=` for which `c @= x` is well-formed, `c @= a` shall also be well-formed and shall have the same value and effects as `c @= x`. The expression `c @= a` shall be constant time and shall be an lvalue referring to `c`.

(5.3) — For every binary operator `@` for which `x @ y` is well-formed, `a @ b` shall also be well-formed and shall have the same value, effects, and value category as `x @ y` provided that value is within the

range of `T`. The expression `a @ b` shall be constant time. If `x @ y` has type `bool`, so too does `a @ b`; if `x @ y` has type `U`, then `a @ b` has type `T`.

6 All integer-like types are explicitly convertible to all integral types and implicitly and explicitly convertible from all integral types.

7 All integer-like types are contextually convertible to `bool` with as if by `a != T(0)`, where `a` is an instance of the integral-like type `T`.

8 For every (possibly *cv*-qualified) integer-like type `T`, `numeric_limits<T>` has been specialized such that:

(7.1) — `numeric_limits<T>::is_specialized` is `true`.

(7.2) — `numeric_limits<T>::is_signed` is `true` if and only if `T` is a signed integer-like type.

(7.3) — `numeric_limits<T>::is_integer` is `true`.

(7.4) — `numeric_limits<T>::digits` is equal to the width of the *`integer-like`* type.

(7.5) — `numeric_limits<T>::digits10` is equal to `static_cast<int> (digits * log10(2))`.

(7.6) — `numeric_limits<T>::min()` and `numeric_limits<T>::max()` return the bounds of `T`'s range, and `numeric_limits<T>::lowest()` returns `numeric_limits<T>::min()`.

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

1 The `WeaklyIncrementable` concept specifies the requirements on types that can be incremented with the pre- and post-increment operators. The increment operations are not required to be equality-preserving, nor is the type required to be `EqualityComparable`.

```
template<class I>
concept WeaklyIncrementable =
  Semiregular<I> &&
  requires(I i) {
    typename iter_difference_t<I>;
-   requires SignedIntegral<iter_difference_t<I>>;
+   requires is-signed-integer-like<iter_difference_t<I>>;
    { ++i } -> Same<I&>; // not required to be equality-preserving
    i++; // not required to be equality-preserving
  };
```

[Editor's note: We want to limit the change to only new iterator concepts, not to the old iterator requirements tables, so change [iterator.iterators]/p2 as follows:]

2 A type `X` satisfies the *Cpp17Iterator* requirements if:

(2.1) - `X` satisfies the *Cpp17CopyConstructible*, *Cpp17CopyAssignable*, and *Cpp17Destructible* requirements (16.5.3.1) and lvalues of type `X` are swappable (16.5.3.2), and

(2.?) — `iterator_traits<X>::difference_type` is a signed integer type or `void`, and

(2.2) - the expressions in Table 73 are valid and have the indicated semantics.

[Editor's note: In section "Header `<ranges>` synopsis" (24.2), immediately following the header synopsis, add a new paragraph as follows:]

1 Within this clause, for some expression `x` of integer-like type (REF), let *`make-unsigned(x)`* be `x` converted to an unsigned integer type of the same width, if any such type exists; otherwise, let it be `x` converted to an unsigned integer-like (REF) type of the same width. For some integer-like type `X`, let *`make-unsigned-t(X)`* be the type of *`make-unsigned(X(0))`*.

[Editor's note: Permit `ranges::size` to return an integer-like type and force it to be unsigned by changing [range.prim.size] as follows:]

24.3.9 `ranges::size`

1 The name `size` denotes a customization point object (16.4.2.2.6). The expression `ranges::size(E)` for some subexpression `E` with type `T` is expression-equivalent to:

(1.1) — *decay-copy*`(extent_v<T>)` if `T` is an array type (6.7.2).

(1.2) — Otherwise, if `disable_sized_range<remove_cv_t<T>>` (24.4.3) is `false`:

(1.2.1) — *decay-copy*`(E.size())` if it is a valid expression and its type `I` ~~models `Integral`~~is integer-like.

(1.2.2) — Otherwise, *decay-copy*`(size(E))` if it is a valid expression and its type `I` ~~models `Integral`~~is integer-like with overload resolution performed in a context that includes the declaration:

```
template<class T> void size(T&&) = delete;
```

and does not include a declaration of `ranges::size`.

(1.3) — Otherwise, `(ranges::end(E) - ranges::begin(E))` if it is a valid expression and the types `I` and `S` of `ranges::begin(E)` and `ranges::end(E)` model `SizedSentinel<S, I>` (23.3.4.8) and `ForwardIterator<I>`. However, `E` is evaluated only once.

(1.4) — Otherwise, `ranges::size(E)` is ill-formed. [*Note:* This case can result in substitution failure when `ranges::size(E)` appears in the immediate context of a template instantiation. — *end note*]

2 [*Note:* Whenever `ranges::size(E)` is a valid expression, its type ~~models `Integral`~~is integer-like. — *end note*]

[Editor's note: Change [range.iter.op.distance]/p3 as follows:]

```
template<Range R>
  constexpr iter_difference_t<iterator_t<R>> ranges::distance(R&& r);
```

3 *Effects:* If `R` models `SizedRange`, equivalent to:

```
-return ranges::size(r); // 24.3.9
+return static_cast<iter_difference_t<iterator_t<R>>>(ranges::size(r)); // 24.3.9
```

Otherwise, equivalent to:

```
return ranges::distance(ranges::begin(r), ranges::end(r)); // 24.3
```

[Editor's note: Change the class synopsis in [range.iota.view] as follows:]

```
template<WeaklyIncrementable W, Semiregular Bound = unreachable_sentinel_t>
  requires weakly-equality-comparable-with<W, Bound>
class iota_view : public view_interface<iota_view<W, Bound>> {
private:
  // 24.6.3.3, class iota_view::iterator
  struct iterator; // exposition only
  // 24.6.3.4, class iota_view::sentinel
  struct sentinel; // exposition only
  W value_ = W(); // exposition only
  Bound bound_ = Bound(); // exposition only
public:
  iota_view() = default;
  constexpr explicit iota_view(W value);
```

```
   constexpr iota_view(type_identity_t<W> value,
                       type_identity_t<Bound> bound);

   constexpr iterator begin() const;
-  constexpr sentinel end() const;
+  constexpr auto end() const;
   constexpr iterator end() const requires Same<W, Bound>;

   constexpr auto size() const
-    requires (Same<W, Bound> && Advanceable<W>) ||
-             (Integral<W> && Integral<Bound>) ||
-             SizedSentinel<Bound, W>
-  { return bound_ - value_; }
+    requires see below;
 };

 template<class W, class Bound>
-  requires (!Integral<W> || !Integral<Bound> || is_signed_v<W> == is_signed_v<Bound>)
+  requires (!is-integer-like<W> || !is-integer-like<Bound> ||
+    (is-signed-integer-like<W> == is-signed-integer-like<Bound>))
 iota_view(W, Bound) -> iota_view<W, Bound>;
```

[Editor's note: Immediately following the `iota_view` class synopsis, insert the following new paragraph and renumber the subsequent paragraphs.]

1 Let *IOTA_DIFF_T(W)* be defined as follows:

- If `W` is not an integral type, or if it is an integral type and `sizeof(iter_difference_t<W>)` is greater than `sizeof(W)`, *IOTA_DIFF_T(W)* is an alias for `iter_difference_t<W>`.
- Otherwise, *IOTA_DIFF_T(W)* is a signed integer type of width not less than the width of `W` plus one if such a type exists.
- Otherwise, *IOTA_DIFF_T(W)* is an unspecified signed integer-like type (REF) of width not less than the width of `W`. [*Note:* It is unspecified whether this type satisfies `WeaklyIncrementable`. — *end note*] [Editor's note: This gives implementors the freedom to use a user-defined type as the difference type without requiring them to do so.]

[Editor's note: Change [range.iota.view]/p4 as follows:]

4 The exposition-only *Advanceable* concept is equivalent to:

```
 template<class I>
 concept Advanceable =
   Decrementable<I> && StrictTtotallyOrdered<I> &&
-  requires(I i, const I j, const iter_difference_t<I> n) {
+  requires(I i, const I j, const IOTA_DIFF_T(I) n) {
     { i += n } -> Same<I&>;
     { i -= n } -> Same<I&>;
-    { j + n } -> Same<I>;
-    { n + j } -> Same<I>;
-    { j - n } -> Same<I>;
+    I(j + n);
+    I(n + j);
+    I(j - n);
```

```
-    { j - j } -> Same<iter_difference_t<I>>;
+    { j - j } -> ConvertibleTo<IOTA_DIFF_T(I)>;
  };
```

Let D be *IOTA_DIFF_T(I)*. Let `a` and `b` be objects of type `I` such that `b` is reachable from `a` after `n` applications of `++a`, for some value `n` of type ~~`iter_difference_t<I>`~~`D`~~, and let D be `iter_difference_t<I>`~~. `I` models *Advanceable* only if

(4.1) — `(a += n)` is equal to `b`.

(4.2) — `addressof(a += n)` is equal to `addressof(a)`.

(4.3) — `I(a + n)` is equal to `(a += n)`.

(4.4) — For any two positive values `x` and `y` of type `D`, if `I(a + D(x + y))` is well-defined, then `I(a + D(x + y))` is equal to `I(I(a + x) + y)`.

(4.5) — `I(a + D(0))` is equal to `a`.

(4.6) — If `I(a + D(n - 1))` is well-defined, then `I(a + n)` is equal to ~~`++(a + D(n - 1))`~~ `[](I c){ return ++c; }(I(a + D(n - 1)))`. [Editor's note: Pre-increment on prvalues is not a requirement of `WeaklyIncrementable`.]

(4.7) — `(b += -n)` is equal to `a`.

(4.8) — `(b -= n)` is equal to `a`.

(4.9) — `addressof(b -= n)` is equal to `addressof(b)`.

(4.10) — `I(b - n)` is equal to `(b -= n)`.

(4.11) — `D(b - a)` is equal to `n`.

(4.12) — `D(a - b)` is equal to `D(-n)`.

(4.13) — `bool(a <= b)` is `true`.

[Editor's note: Include the PR of https://github.com/ericniebler/stl2/issues/612 here.]

```
constexpr iota_view(type_identity_t<W> value, type_identity_t<Bound> bound);
```

7 *Expects:* Bound denotes `unreachable_sentinel_t` or `bound` is reachable from `value`. When `StrictTotallyOrderedWith<W, Bound>` is satisfied, then `value <= bound` is `true`.

8 *Effects:* Initializes `value_` with `value` and `bound_` with `bound`.

[Editor's note: Drive-by fix for https://github.com/ericniebler/stl2/issues/615. Change [range.iota.view]/10 as follows:]

```
-constexpr sentinel end() const;
+constexpr auto end() const;
```

10 *Effects:* Equivalent to~~: `return sentinel{bound_};`~~

```
if constexpr (Same<Bound, unreachable_sentinel_t>)
  return unreachable_sentinel;
else
  return sentinel{bound_};
```

[Editor's note: After [range.iota.view]/11, insert a new paragraph 12 as follows:]

```
constexpr auto size() const requires see below;
```

12 *Remarks:* The expression in the *requires-clause* is equivalent to

```
(Same<W, Bound> && Advanceable<W>) || (Integral<W> && Integral<Bound>) ||
SizedSentinel<Bound, W>
```

13 *Effects:* Equivalent to

```
if constexpr (is-integer-like<W>)
  return (value_ < 0)
    ? ((bound_ < 0)
      ? make-unsigned(-value_) - make-unsigned(-bound_)
      : make-unsigned(bound_) + make-unsigned(-value_))
    : make-unsigned(bound_) - make-unsigned(value_);
else
  return make-unsigned(bound_ - value_);
```

[Editor's note: Change the class synopsis of `iota_view::iterator` in [range.iota.iterator] as follows:]

```
template<class W, class Bound>
struct iota_view<W, Bound>::iterator {
private:
  W value_ = W(); // exposition only
public:
  using iterator_category = see below ;
  using value_type = W;
-  using difference_type = iter_difference_t<W>;
+  using difference_type = IOTA_DIFF_T(W);

  iterator() = default
...
```

[Editor's note: Change `iota_view::iterator::operator+=` in [range.iota.iterator]/11 as follows:]

```
constexpr iterator& operator+=(difference_type n)
  requires Advanceable<W>;
```

11 *Effects:* Equivalent to:

```
-value_ += n;
+if constexpr (is-integer-like<W> && !is-signed-integer-like<W>) {
+  if (n >= difference_type(0))
+    value_ += static_cast<W>(n);
+  else
+    value_ -= static_cast<W>(-n);
+} else {
+  value_ += n;
+}
 return *this;
```

[Editor's note: Change `iota_view::iterator::operator-=` in [range.iota.iterator]/12 as follows:]

```
constexpr iterator& operator-=(difference_type n)
  requires Advanceable<W>;
```

11 *Effects:* Equivalent to:

```
-value_ -= n;
+if constexpr (is-integer-like<W> && !is-signed-integer-like<W>) {
+  if (n >= difference_type(0))
+    value_ -= static_cast<W>(n);
+  else
+    value_ += static_cast<W>(-n);
+} else {
+  value_ -= n;
+}
 return *this;
```

[Editor's note: Change `iota_view::iterator::operator[]` in [range.iota.iterator]/13 as follows:]

```
constexpr W operator[](difference_type n) const
  requires Advanceable<W>;
```

13 *Effects:* Equivalent to: `return W(value_ + n);`

[Editor's note: Change `iota_view::iterator::operator+` in [range.iota.iterator]/20 as follows:]

```
friend constexpr iterator operator+(iterator i, difference_type n)
  requires Advanceable<W>;
```

20 *Effects:* Equivalent to: `return ~~iterator{i.value_ + n}~~ i += n;`

[Editor's note: Change `iota_view::iterator::operator-` in [range.iota.iterator]/22 as follows:]

```
friend constexpr iterator operator-(iterator i, difference_type n)
  requires Advanceable<W>;
```

22 *Effects:* Equivalent to: `~~i + -n~~ i -= n;`

[Editor's note: Change `iota_view::iterator::operator-` in [range.iota.iterator]/22 as follows:]

```
friend constexpr difference_type operator-(const iterator& x, const iterator& y)
  requires Advanceable<W>;
```

22 *Effects:* Equivalent to~~: return x.value_ - y.value_;~~

```
using D = difference_type;
if constexpr (is-integer-like<W>) {
  if constexpr (is-signed-integer-like<W>)
    return D(D(x.value_) - D(y.value_));
  else
    return (y.value_ > x.value_)
      ? D(-D(y.value_ - x.value_))
      : D(x.value_ - y.value_);
} else {
   return x.value_ - y.value_;
}
```

[Editor's note: Drive-by fix of https://github.com/ericniebler/stl2/issues/613. Change the synopsis for class `iota_view::sentinel` as follows:]

```
template<class W, class Bound>
struct iota_view<W, Bound>::sentinel {
private:
  Bound bound_ = Bound(); // exposition only
public:
  sentinel() = default;
  constexpr explicit sentinel(Bound bound);

  friend constexpr bool operator==(const iterator& x, const sentinel& y);
  friend constexpr bool operator==(const sentinel& x, const iterator& y);
  friend constexpr bool operator!=(const iterator& x, const sentinel& y);
  friend constexpr bool operator!=(const sentinel& x, const iterator& y);

+ friend constexpr iter_difference_t<W> operator-(const iterator& x, const sentinel& y)
+   requires SizedSentinel<Bound, W>;
+ friend constexpr iter_difference_t<W> operator-(const sentinel& x, const iterator& y)
+   requires SizedSentinel<Bound, W>;
};
```

[Editor's note: After [range.iota.sentinel]/5, add the following new paragraphs:]

```
friend constexpr iter_difference_t<W> operator-(const iterator& x, const sentinel& y)
  requires SizedSentinel<Bound, W>;
```

6 *Effects:* Equivalent to: `return x.value_ - y.bound_;`

```
friend constexpr iter_difference_t<W> operator-(const sentinel& x, const iterator& y)
  requires SizedSentinel<Bound, W>;
```

7 *Effects:* Equivalent to: `return y - x;`

6 Acknowledgements

Many thanks to Casey Carter for his willingness to discuss this issue at length with me and entertain my many half-baked ideas for addressing it over the past year, and also for reviewing this paper in detail.

7 References

[range-v3] Range-v3: Ranges library for C++14/17/2a.
https://github.com/ericniebler/range-v3