Views and Size Types

Document #:	P1523R1
Date:	2019-07-19
Project:	Programming Language C++ Library Evolution Working Group Library Working Group

Reply-to:	Eric Niebler <eric.niebler@gmail.com>

1 Introduction

In the Rapperswil '18 meeting, after protracted discussion, it was decided that the `span` type's `.size()` method should be changed to return an unsigned integer type. The primary motivation for the change was consistency with the rest of the Standary Library, which uses unsigned integers for size types exclusively.

Although not discussed then, the new `View` types in the Ranges clause also need to be changed to have unsigned size types in order to keep the Standard Library consistent.

This proposed resolution in this paper assumes the adoption of its companion paper [P1522R0], "Iterator Difference Type and Integer Overflow". Should that paper not be adopted, the resolution proposed in this paper can be trivially adapted to apply directly to the working draft.

2 Design Considerations

Should the `subrange(I i, S s, iter_difference_t<I> n)` constructor be changed to take an unsigned type, like `span`?

3 Proposed Resolution

[Editor's note: Change [range.prim.size] as follows (edits relative to [P1522R0]):]

[Editor's note: If the user has defined a `size` function (either a member function or a free function) that returns a signed integer, this does *not* force it to be unsigned. Presumably, the user knows what they are doing. Rather, only force the size to be unsigned if they have not specified a `size` function, and we are using `distance` on `SizedSentinel<S, I>` to compute the size, which yields a signed integer always.]

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 is integer-like.

1.2.2 — Otherwise, *decay-copy*(size(E)) if it is a valid expression and its type I 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, *make-unsigned-like*(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 is integer-like. — *end note*]

[Editor's note: Change the class declaration of subrange ([range.subrange]/p1) as follows:]

```
 [...]

 template<Iterator I, Sentinel<I> S = I, subrange_kind K =
     SizedSentinel<S, I> ? subrange_kind::sized : subrange_kind::unsized>
   requires (K == subrange_kind::sized || !SizedSentinel<S, I>)
 class subrange : public view_interface<subrange<I, S, K>> {
 private:
   static constexpr bool StoreSize = // exposition only
     K == subrange_kind::sized && !SizedSentinel<S, I>;
   I begin_ = I(); // exposition only
   S end_ = S(); // exposition only
-   iter_difference_t<I> size_ = 0; // exposition only; present only
+   make-unsigned-like-t(iter_difference_t<I>) size_ = 0; // exposition only; present only
                                                          // when StoreSize is true
public:
   subrange() = default;

   constexpr subrange(I i, S s) requires (!StoreSize);

-   constexpr subrange(I i, S s, iter_difference_t<I> n)
+   constexpr subrange(I i, S s, make-unsigned-like-t(iter_difference_t<I>) n)
     requires (K == subrange_kind::sized);

   template<not-same-as<subrange> R>
     requires forwarding-range<R> &&
       ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
   constexpr subrange(R&& r) requires (!StoreSize || SizedRange<R>);

   template<forwarding-range R>
     requires ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
-   constexpr subrange(R&& r, iter_difference_t<I> n)
+   constexpr subrange(R&& r, make-unsigned-like-t(iter_difference_t<I>) n)
     requires (K == subrange_kind::sized)
       : subrange{ranges::begin(r), ranges::end(r), n}
```

```
   {}

   template<not-same-as<subrange> PairLike>
     requires pair-like-convertible-to<PairLike, I, S>
   constexpr subrange(PairLike&& r) requires (!StoreSize)
     : subrange{std::get<0>(std::forward<PairLike>(r)),
                std::get<1>(std::forward<PairLike>(r))}
   {}

   template<pair-like-convertible-to<I, S> PairLike>
-   constexpr subrange(PairLike&& r, iter_difference_t<I> n)
+   constexpr subrange(PairLike&& r, make-unsigned-like-t(iter_difference_t<I>) n)
     requires (K == subrange_kind::sized)
     : subrange{std::get<0>(std::forward<PairLike>(r)),
                std::get<1>(std::forward<PairLike>(r)), n}
   {}

 [...]
   constexpr bool empty() const;
-  constexpr iter_difference_t<I> size() const
+  constexpr make-unsigned-like-t(iter_difference_t<I>) size() const
     requires (K == subrange_kind::sized);
 [...]
 };

 template<Iterator I, Sentinel<I> S>
- subrange(I, S, iter_difference_t<I>) -> subrange<I, S, subrange_kind::sized>;
+ subrange(I, S, make-unsigned-like-t(iter_difference_t<I>)) ->
+   subrange<I, S, subrange_kind::sized>;

 template<iterator-sentinel-pair P>
 subrange(P) -> subrange<tuple_element_t<0, P>, tuple_element_t<1, P>>;

 template<iterator-sentinel-pair P>
- subrange(P, iter_difference_t<tuple_element_t<0, P>>) ->
+ subrange(P, make-unsigned-like-t(iter_difference_t<tuple_element_t<0, P>>)) ->
   subrange<tuple_element_t<0, P>, tuple_element_t<1, P>, subrange_kind::sized>;

 template<forwarding-range R>
 subrange(R&&) ->
   subrange<iterator_t<R>, sentinel_t<R>,
     (SizedRange<R> || SizedSentinel<sentinel_t<R>, iterator_t<R>>)
   ? subrange_kind::sized : subrange_kind::unsized>;

 template<forwarding-range R>
- subrange(R&&, iter_difference_t<iterator_t<R>>) ->
+ subrange(R&&, make-unsigned-like-t(iter_difference_t<iterator_t<R>>)) ->
   subrange<iterator_t<R>, sentinel_t<R>, subrange_kind::sized>;

 template<size_t N, class I, class S, subrange_kind K>
   requires (N < 2)
```

```
constexpr auto get(const subrange<I, S, K>& r);
```

[Editor's note: Change [range.subrange.ctor]/p3-5 as follows:]

[...]

```
-constexpr subrange(I i, S s, iter_difference_t<I> n)
+constexpr subrange(I i, S s, make-unsigned-like-t(iter_difference_t<I>) n)
   requires (K == subrange_kind::sized);
```

3 Expects: `[i, s)` is a valid range, and `n == make-unsigned-like(ranges::distance(i, s))`.

4 Effects: Initializes `begin_` with `i` and `end_` with `s`. If `StoreSize` is `true`, initializes `size_` with `n`.

5 [*Note:* Accepting the length of the range and storing it to later return from `size()` enables `subrange` to model `SizedRange` even when it stores an iterator and sentinel that do not model `SizedSentinel`. — *end note*]

[Editor's note: Change [range.subrange.access]/p4 as follows:]

```
-constexpr iter_difference_t<I> size() const
+constexpr make-unsigned-like-t(iter_difference_t<I>) size() const
   requires (K == subrange_kind::sized);
```

4 *Effects:*

4.1 — If `StoreSize` is `true`, equivalent to: `return size_;`

4.2 — Otherwise, equivalent to: `return make-unsigned-like( end_ - begin_ ) ;`

[Editor's note: Change [range.subrange.access]/p8 as follows:]

```
constexpr subrange& advance(iter_difference_t<I> n);
```

8 Effects: Equivalent to:

8.1 If `StoreSize` is `true`,

```
-size_ -= n - ranges::advance(begin_, n, end_);
+auto d = n - ranges::advance(begin_, n, end_);
+if (d >= 0)
+  size_ -= make-unsigned-like(d);
+else
+  size_ += make-unsigned-like(-d);
return *this;
```

8.2 Otherwise,

```
ranges::advance(begin_, n, end_);
return *this;
```

[Editor's note: Change the class declaration of `range::empty_view` in [range.empty.view] as follows:]

```
namespace std::ranges {
  template<class T>
    requires is_object_v<T>
```

```
   class empty_view : public view_interface<empty_view<T>> {
   public:
     static constexpr T* begin() noexcept { return nullptr; }
     static constexpr T* end() noexcept { return nullptr; }
     static constexpr T* data() noexcept { return nullptr; }
-    static constexpr ptrdiff_t size() noexcept { return 0; }
+    static constexpr size_t size() noexcept { return 0; }
     static constexpr bool empty() noexcept { return true; }

     friend constexpr T* begin(empty_view) noexcept { return nullptr; }
     friend constexpr T* end(empty_view) noexcept { return nullptr; }
   };
 }
```

[Editor's note: Change the class declaration of `range::single_view` in [range.single.view] as follows:]

```
 namespace std::ranges {
   template<CopyConstructible T>
     requires is_object_v<T>
   class single_view : public view_interface<single_view<T>> {
   private:
     semiregular<T> value_; // exposition only
   public:
     single_view() = default;
     constexpr explicit single_view(const T& t);
     constexpr explicit single_view(T&& t);
     template<class... Args>
       requires Constructible<T, Args...>
     constexpr single_view(in_place_t, Args&&... args);
     constexpr T* begin() noexcept;
     constexpr const T* begin() const noexcept;
     constexpr T* end() noexcept;
     constexpr const T* end() const noexcept;
-    static constexpr ptrdiff_t size() noexcept;
+    static constexpr size_t size() noexcept;
     constexpr T* data() noexcept;
     constexpr const T* data() const noexcept;
   };
 }
```

[Editor's note: Change [range.single.view]/p6 as follows:]

```
-static constexpr ptrdiff_t size() noexcept;
+static constexpr size_t size() noexcept;
```

6 *Effects:* Equivalent to: return 1;

4 References

[P1522R0] Eric Niebler. 2019. P1522R0: Iterator Difference Type and Integer Overflow. http://wg21.link/P1522R0