Helpful pointers for ContiguousIterator

1 Abstract

The support for contiguous iterators in the working draft is missing a useful feature: a mechanism to convert a contiguous iterator into a pointer that denotes the same object. This paper proposes that std::to_address be that mechanism.

Table 1 — Tony Table

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern &quot;C&quot; int some_c_api(T* ptr, size_t size); template&lt;ContiguousIterator I&gt; int try_useful_things(I i, size_t n) { // Expects: [i, n) is a valid range if (n == 0) { // Oops - can’t dereference past-the-end iterator throw something; } return some_c_api(addressof(*i), n); } template&lt;ContiguousIterator I&gt; int try_useful_things(I i, I j) { // Expects: [i, j) is a valid range if (i == j) { // Oops - can’t dereference past-the-end iterator throw something; } return some_c_api(addressof(*i), addressof(*i) + (j - i)); }</td>
<td>extern &quot;C&quot; int some_c_api(T* ptr, size_t size); template&lt;ContiguousIterator I&gt; int try_useful_things(I i, size_t n) { // Expects: [i, n) is a valid range return some_c_api(to_address(i), n); } template&lt;ContiguousIterator I&gt; int try_useful_things(I i, I j) { // Expects: [i, j) is a valid range return some_c_api(to_address(i), to_address(j)); }</td>
</tr>
</tbody>
</table>

1.1 Revision History

1.1.1 Revision 0

— Initial revision.

2 Problem description

P0944R0 “Contiguous ranges” [1] proposed support for contiguous ranges and iterators, which was merged into P0896R4 “The One Ranges Proposal” [4] and then merged into the Working Draft. Neither P0944R0 nor P0896R4 proposed a means of obtaining a pointer to the element denoted by an arbitrary ContiguousIterator. At the time, the author was under the impression that such a mechanism had been a “third rail” for past contiguous iterator proposals [3], and that requiring such a mechanism would make it impossible to require
the iterators of the Standard Library containers to model `ContiguousIterator`. Those implementability concerns have since been rectified.

Note that obtaining a pointer value from a dereferenceable `ContiguousIterator` is trivial: `std::addressof(*i)` returns such a pointer value for a contiguous iterator `i`. Dereferencing a non-dereferenceable iterator is (unsurprisingly) not well-defined, so this mechanism isn’t suitable for iterators not known to be dereferenceable. Obtaining a pointer value for the potentially non-dereferenceable iterator `j` that is the past-the-end iterator of a range `[i, j)` thus requires a different mechanism that is well-defined for past-the-end iterators. Ideally the mechanism would also be well-defined for dereferenceable iterators so it can be used uniformly.

P0653R2 “Utility to convert a pointer to a raw pointer” [2] added the function `std::to_address` ([pointer.conversion]) to the Standard Library which converts values of so-called “fancy” pointer types and standard smart pointer types to pointer values. In the interest of spelling similar things similarly, it seems a good idea to reuse this facility to convert `ContiguousIterator` to pointer values. In practice, that means that a type `I` must be a pointer type or

— specialize `pointer_traits<I>`, and

— implement `pointer_traits<I>::to_address`, or admit past-the-end (potentially non-dereferenceable) iterator values in `operator->()`.

### 3 Proposal

The basic proposal is to add a requirement to the `ContiguousIterator` concept that the expression `std::to_address(i)` for an lvalue `i` of type `const I` must

— be well-formed and yield a pointer of type `add_pointer_t<iter_reference_t<I>>`,

— be well-defined for both dereferenceable and past-the-end pointer values,

— yield a pointer value equal to `std::addressof(*i)` if `i` is dereferenceable, or equal to `std::addressof(*(i - 1))` if `i - 1` is dereferenceable.

Since dereferenceable `ContiguousIterator`s always denote objects - their reference types are always lvalue references - they can always feasibly implement the `->` operator. `->` is useful in contexts where the value type of the iterator is concrete so we propose requiring it for all `ContiguousIterator`s. [Note: Recall that the iterator concepts do not generally require `operator->` as do the “old” iterator requirements. —end note]

Now that there’s a mechanism to retrieve a pointer from a potentially non-dereferenceable iterator, we can also cleanup the edge cases in `ranges::data` and `ranges::view_interface::data` which return `nullptr` for an empty `ContiguousRange` rather than unconditionally returning the pointer value that the `begin` iterator denotes.

### 4 Technical specifications

Change the `[iterator.concept.contiguous]` as follows:

```cpp
template<class I>
concept ContiguousIterator =
    RandomAccessIterator<I> &&
    DerivedFrom<ITER_CONCEPT(I), contiguous_iterator_tag> &&
    is_lvalue_reference_v<iter_reference_t<I>> &&
    Same<iter_value_t<I>, remove_cvref_t<iter_reference_t<I>>>&&
    requires(const I& i) { 
        to_address(i) -> Same<add_pointer_t<iter_reference_t<I>>, iter_value_t<I>>;
    } &&
    (is_pointer_v<I> || requires(const I& i) { 
        i.operator->(); } -> Same<add_pointer_t<iter_reference_t<I>>, iter_value_t<I>>;
    });
```

2 Let `a` and `b` be dereferenceable iterators and `c` a non-dereferenceable iterator of type `I` such that `b` is reachable from `a` and `c` is reachable from `b`, and let `D` be `iter_difference_t<I>`. The type `I`
models **ContiguousIterator** only if \( \text{addressof}(\ast(a + D(b - a))) \) is equal to \( \text{addressof}(\ast a + D(b - a)) \).

\[\begin{align*}
(2.1) & \quad \text{to_address}(b) == \text{to_address}(a) + D(b - a), \\
(2.2) & \quad \text{to_address}(c) == \text{to_address}(a) + D(c - a), \text{and} \\
(2.3) & \quad \text{if } I \text{ is not a pointer type, } a.\text{operator->}() == \text{to_address}(a).
\end{align*}\]

Change [range.prim.data] as follows:

1. The name `data` denotes a customization point object ([customization.point.object]). The expression `ranges::data(E)` for some subexpression `E` is expression-equivalent to:

\[\begin{align*}
(1.1) & \quad \text{If } E \text{ is an lvalue, } \text{DECAY_COPY}(E.\text{data}()) \text{ if it is a valid expression of pointer to object type.} \\
(1.2) & \quad \text{Otherwise, if } \text{ranges::begin}(E) \text{ is a valid expression whose type models } \text{ContiguousIterator}, \\
& \quad \quad \text{to_address(\text{ranges::begin}(E))} \\
& \quad \quad \text{ranges::begin}(E) == \text{ranges::end}(E) ? \text{nullptr} : \text{addressof}(\ast\text{ranges::begin}(E))
\end{align*}\]

except that `E` is evaluated only once.

\[\begin{align*}
(1.3) & \quad \text{Otherwise, } \text{ranges::data}(E) \text{ is ill-formed.}
\end{align*}\]

Change [view.interface] as follows:

```cpp
namespace std::ranges {
    ...

template<class D>
    requires is_class_v<D> && Same<D, remove_cv_t<D>>
    class view_interface : public view_base {
        ...

    constexpr auto data() requires ContiguousIterator<iterator_t<D>> { 
        return ranges::empty(derived()) ? nullptr : addressof(*ranges::begin(derived()));
        return to_address(ranges::begin(derived()));
    }

    constexpr auto data() const
        requires Range<Const D> && ContiguousIterator<iterator_t<Const D>> { 
        return ranges::empty(derived()) ? nullptr : addressof(*ranges::begin(derived()));
        return to_address(ranges::begin(derived()));
    }
}
```

**Bibliography**


