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 \texttt{std::to_address} be that mechanism.

Table 1 — Tony Table

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>extern &quot;C&quot; int some_c_api(T* ptr, size_t size);</code></td>
<td><code>extern &quot;C&quot; int some_c_api(T* ptr, size_t size);</code></td>
</tr>
<tr>
<td><code>extern &quot;C&quot; int other_c_api(T* first, T* last);</code></td>
<td><code>extern &quot;C&quot; int other_c_api(T* first, T* last);</code></td>
</tr>
</tbody>
</table>

```cpp
template<ContiguousIterator I>
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<ContiguousIterator I>
int try_useful_things(I i, I j) {
    // Expects: [i, j) is a valid range
    if (j == i) {
        // Oops - can't dereference past-the-end iterator
        throw something;
    }
    return other_c_api(addressof(*i), addressof(*i) + (j - i));
}
```

```cpp
template<ContiguousIterator I>
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<ContiguousIterator I>
int try_useful_things(I i, I j) {
    // Expects: [i, j) is a valid range
    return other_c_api(to_address(i), to_address(j));
}
```

1.1 Revision History

1.1.1 Revision 1

— Update Tony Table: C APIs can’t be overloaded, and add a bit of markup to make the differences stand out.
— Correct bad pointer arithmetic in the description of the address of a past-the-end iterator whose predecessor is dereferenceable.
— Remove bad \texttt{;} after expression in a \texttt{compound-condition} in the definition of \texttt{ContiguousIterator}.
— Remove \texttt{operator->} requirement (which was not a core part of the proposal) due to LWG concerns.
1.1.2 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” [2] 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 well-defined 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 ContiguousIterators to pointer values. In practice, that means that a type I must be a pointer type or

— specialize pointer_traits<I> with a member element_type or have a nested member element_type so instantiation of pointer_traits<I> succeeds, and

— Either 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 1 + std::addressof(*(i - 1)) if i - 1 is dereferenceable.

Since dereferenceable ContiguousIterators 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 ContiguousIterators. [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 [iterator.concept.contiguous] as follows:
template<class I>
concept ContiguousIterator =
  RandomAccessIterator<I> &&
  Requires<IterConcept(I), contiguous_iterator_tag> &&
  IsLvalueReferenceV<IterReferenceT<I>> &&
  Same<IterValueT<I>, RemoveCvRefT<IterReferenceT<I>>> &&
  Requires(const I& i) { 
    to_address(i) -> Same<AddPointerT<IterReferenceT<I>>>; 
  };

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 addressof(*(a + D(b - a))) is equal to addressof(*a) + D(b - a).

(2.1) to_address(a) == addressof(*a),
(2.2) to_address(b) == to_address(a) + D(b - a), and
(2.3) to_address(c) == to_address(a) + D(c - a).

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:

(1.1) If E is an lvalue, decay-copy(E.data()) if it is a valid expression of pointer to object type.
(1.2) Otherwise, if ranges::begin(E) is a valid expression whose type models ContiguousIterator,

    ranges::begin(E) == ranges::end(E) ? nullptr : addressof(*ranges::begin(E))

    except that E is evaluated only once.
(1.3) Otherwise, ranges::data(E) is ill-formed. [Note: This case can result in substitution failure when ranges::data(E) appears in the immediate context of a template instantiation. —end note]

Change [view.interface] as follows:

namespace std::ranges {
  template<class D>
  requires is_class_v<D> && Same, 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
