Contiguous Iterators: Pointer Conversion & Type Trait

Document number: N4183
Date: 2014-10-10
Project: Programming Language C++, Library Evolution Working Group
Replaces: n3884: Contiguous Iterators: A Refinement of Random Access Iterators
Reply-to: Nevin "☺" Liber mailto:nliber@drw.com

Introduction

This is a proposal to:

- Add a mechanism for converting a contiguous iterator to a pointer
- A type trait for contiguous iterators

It builds upon the discussion in Issaquah on the previous version of this paper (n3884) and is dependent on n4132 Contiguous Iterators by Jens Maurer.

Changes from n3884:

N4132 covers the wording changes for the contiguous data structures (basic_string, array, vector and valarray) in the standard, so those wording changes are removed from this proposal, greatly simplifying it in the process.

While contiguous iterators are a refinement of random access iterators, LEWG was split on whether or not it would break too much code to derive an contiguous_iterator_tag from random_access_iterator_tag and change existing iterator_traits to use that tag. This is because some code in the wild assumes that random_access_iterator_tag is at the bottom of the hierarchy and checks for that exact tag instead of doing tag dispatching via overloading. Having given this a bit of thought, I agree that potential code breakage is too much. To avoid this problem, a new, independent type trait is being proposed.

The preferred mechanism for a std::pointer_from(i) to convert a contiguous iterator into a pointer (without requiring any external information, such as the container it comes from) is to use ADL to perform an unqualified call to do_pointer_from(i) (names still to be bike shedded, of course).

Converting a contiguous iterator to a raw pointer via do_pointer_from()/std::pointer_from()

It is highly desirable to be able to convert a contiguous iterator to a raw pointer without requiring any external information (such as the container it comes from). Some use cases:

- Passing buffers to C APIs.
- Algorithm improvements; e.g., memcpy a contiguous POD.
- Classes such as string_view and array_view can be constructed from a contiguous range.
- The Boost.Container library uses this functionality (the function is iterator_to_raw_pointer, although that may not be limited to contiguous iterators)

We need a function because the construct std::addressof(*i) is only valid if i is dereferenceable, and we wish to convert non-dereferenceable end-of-range iterators to raw pointers as well. When such a function is not provided, we have seen people use the construct &*i, even though that results in undefined behavior for end-of-range iterators.

Because pointers themselves can be contiguous iterators, it would be impossible to require that such a conversion be done via a member function.

The proposed interface is a free function std::pointer_from which uses argument dependent lookup to call an unqualified do_pointer_from for conversion from an iterator to a pointer. This gives container implementers the most flexibility.

A potential downside to this method is that it may not be easy (or possible) to extend pointer_from() to other types (non-contiguous iterators, smart pointers, etc.). iterator_to_raw_pointer() in Boost.Container shows that such a function can be useful to other iterators. There is also boost::get_pointer(), which shows that such a function can be useful to things like smart pointers.

Add the following to [iterator.synopsis]:

```
// contiguous iterators:
template<class T>
constexpr T* do_pointer_from(T* p) noexcept;

template<class ContiguousIterator>
constexpr
auto pointer_from(ContiguousIterator i)
noexcept(noexcept(do_pointer_from(i)))
-> decltype(do_pointer_from(i));
```

Add the following to [contiguous.iterators]:

Table 1 - Contiguous iterator requirements (in addition to random access iterator)

Expression	Return Type	Operational semantics	Assertion/note pre-post-condition
`do_pointer_from(r)`	`T*`	If `r` is dereferenceable, `std::addressof(*r)`. If the expression `r-1` is valid, `std::addressof(*(r-1))+1`.	`do_pointer_from` is looked up in the associated namespace [basic.lookup.argdep]. `do_pointer_from(r)` returns a valid pointer.
`a == b`			pre: `(a,b)` is in the domain of `==`. `do_pointer_from(a) == do_pointer_from(b)`.

Contiguous iterator operations [contiguous.iterator.operations]

```
template<class T>
constexpr T* do_pointer_from(T* p) noexcept;
```

Effects: returns p.

```
template<class ContiguousIterator>
constexpr
auto pointer_from(ContiguousIterator i)
noexcept(noexcept(do_pointer_from(i)))
-> decltype(do_pointer_from(i));
```

Effects: Returns a valid pointer. If `i` is dereferenceable, returns `std::addressof(*i)`. If the expression `i-1` is valid, returns `std::addressof(*(i-1))+1`. [Note: For a valid iterator range `[a,b)` with dereferenceable `a`, the corresponding range denoted by pointers is `[pointer_from(*a),pointer_from(*a) + (b-a))`; `b` might not be dereferenceable. – end note]

Drafting note: Please check that the above wording is both correct and sufficient to cover dereferenceable iterators, end-of-range iterators and empty-range iterators.

Add the following to Annex C (informative) Compatibility:

Code which defines the function `do_pointer_from(T)` in their own namespace may cause breakage.

Known open issues:

1. Is wording needed for the contiguous containers (array, basic_string, vector, valarray)? I believe the contiguous iterator requirements section is sufficient. However, if it is not, such wording would have to take into account that these containers could use raw pointers for iterators or share iterators across container types.
2. Bike shedding for std::pointer_from and do_pointer_from. Previous suggestions:

Call	Define
std::pointer_from	do_pointer_from
std::adl_pointer_from	pointer_from
std::pointer_from	std_do_pointer_from
std::get_pointer	prefixed
std::as_pointer	
std::pointer_from_iterator	
std::to_pointer	

Prefixed for ADL
do_
std_do_
adl_
customized_
custom_

Alternate to ADL

Jens Maurer suggests we consider an alternate to ADL lookup; namely, add pointer_from as a static member function in the iterator_traits class: *I appreciate the extensibility and configurability aspects of it, and I certainly value the flexibility in something like hash_append(), but it seems somewhat over-the-top to use this for rather arcane functionality such as pointer_from(). Can we put that as a static member into std::iterator_traits<> instead?*

My position is that I am less concerned with the actual mechanism as long as such a mechanism exists.

The plus side is that this would be a backwards compatible change to iterator_traits; it doesn't interfere with users that have already specialized it for their own iterators. The main downside is that if someone adds this function for a non-contiguous iterator, any type trait for contiguous iterator based on its presence/ absence would be incorrect.

contiguous_iterator_tag

Jens Maurer would like us to revisit publicly deriving a contiguous_iterator_tag from random_access_iterator_tag and updating the appropriate sections of the standard to use it. *Do we ever expect to have any that are not random-access? If not, I'd really like to extend the iterator tag hierarchy. It seems the code breakage will be loud (not silent),so people will notice and fix their code when moving to C ++1y. We'll be stuck with any hack we apply here until the end of time.*

As I wrote in n3884: *Given that contiguous iterators meet all the requirements of random access iterators, it made sense to publicly derive contiguous_iterator_tag from random_access_iterator_tag. This has the added benefit for those people who use tag dispatching on iterator categories that their code still "just works". Note: this may break or produce sub-optimal performance for existing code that specifically looks for a random_access_iterator_tag (such as in a template specialization), but the Standard has been quite clear that the proper way to use these is by tag dispatching (n3797 24.4.3 [std.iterator.tags] for examples on how to do so), so such breakage in practice should be minimal.*

It has since been brought up that iterator wrappers (such as reverse_iterator, skipping iterators, etc.) tend to just copy verbatim the iterator category of the underlying iterator. That code would break, and perhaps not noisily at compile time. While we can easily fix the ones in the standard (n3884 proposed to do that for std::reverse_iterator, for instance), these wrappers appear in existing code bases (such as boost::reverse_iterator) and the risk of breakage is too great.

Also, there has been talk about revamping the iterator categories (such as suggested in n1550), and if that comes to pass, would likely be a better time to introduce a contiguous_traversal_tag.

Type trait

There are three design choices:
- Make it like other type traits such as is_pointer, where it has exactly one responsibility.
- Make it a superset/replacement of the current iterator_traits, and either refine the iterator_category there or add a new type/value to indicate contiguousness. The big disadvantage to this is that users have to specialize two separate classes in two different ways (making it error-prone) when they add their own contiguous containers, as well as error-prone in usage (because the two classes would be subtly different)
- Do something like allocator_traits that can wrap an iterator.

Because of the error-proneness of the second option, the author prefers the first option and will explore that.

is_contiguous_iterator trait:

Add the following to [meta.type.synop]:

```
// contiguous iterator properties:
template <class T> struct is_contiguous_iterator;
```

Add the following to Table 49 – Type property predicates:

Template	Condition	Preconditions

`template <class T>`
`struct`
`is_contiguous_iterator`
`;`
	`T` is a contiguous iterator	

Drafting note: is the above wording sufficient, given the requiresments on contiguous iterators?

Open issues:

1. Does is_contiguous_iterator belong in <type_traits>?
2. Does is_contiguous_iterator belong in <iterator>?
3. Should is_contiguous_iterator be available when either <type_traits> or <iterator> is included?
4. If pointer_from() is a static member function, does it belong in this type trait instead of and/or in addition to iterator_traits?
5. Bike shedding is_contiguous_iterator.

Impact on the Standard

All of the text is a pure addition to the standard. The only caveat is if we use ADL we are reserving the name do_pointer_from in user namespaces.

Sample implementation:

```
namespace std
{
    template<typename T>
    constexpr T* do_pointer_from(T* p) noexcept { return p; }

    template<typename ContiguousIterator>
    constexpr
    auto pointer_from(ContiguousIterator i)
    noexcept(noexcept(do_pointer_from(i)))
    -> decltype(do_pointer_from(i))         // necessary for SFINAE
    { return do_pointer_from(i); }

    namespace detail
    {
        template<typename, typename = void>
        struct contiguous_iterator_impl
        : std::false_type
        {};

        // Uses void_t from n3911
        template<typename I>
        struct contiguous_iterator_impl<I,
        std::void_t<decltype(std::pointer_from(std::declval<I>()))>>
        : std::true_type
        {};
    } // detail namespace

    template<typename I>
    struct is_contiguous_iterator = typename detail::contiguous_iterator_impl<I>::type;

} // std namespace
```

Acknowledgements

Thanks to Beman Dawes for suggesting that I propose this.

Thanks to Jeffrey Yasskin for both string_view and suggesting that contiguous iterators be convertible to pointers.

Thanks to Stephan T. Lavavej for pushing the ADL solution for converting pointers.

Special thanks to Tom Rodgers for presenting n3884 in Issaquah.

Thanks to Jens Maurer for n4132, which vastly simplifies this paper, as well as his many comments.

Thanks to Ion Gaztañaga for pointing out that the Boost.Container library has iterator_to_raw_pointer().

Thanks to Walter Brown for n3911, as I use it in my sample implementation.

Thanks to Eric Niebler, Dave Abrahams, Gabriel Dos Reis, Howard Hinnant, P.J. Plauger, Sean Parent, Andrew Koenig, Bjarne Stroustrup, Nikolay Ivchenkov, Herb Sutter, Matt Austern, Tony Van Eerd, Mathias Gaunard, Olaf van der Spek, Anthony Syzdek, Matt Godbolt, Marc Glisse, Jonathan Wakely and Richard Smith for discussions on contiguous iterators on various mailing lists. (If I missed anyone, please let me know.)

Thank them / blame me for things you like and don't like, respectively.

References

N3884, Contiguous Iterators: A Refinement of Random Access Iterators, Nevin "☺" Liber

N4132, Contiguous Iterators, Jens Maurer

N1550, New Iterator Concepts, David Abrahams, Jeremy Siek, Thomas Witt

N3911, TransformationTrait Alias void_t, Walter E. Brown

N3936, Working Draft, Standard for Programming Language C++

N4081, Working Draft, C++ Extensions for Library Fundamentals

N4087, Multidimensional bounds, index and array_view revision 3, Łukasz Mendakiewicz & Herb Sutter

Boost.Container, iterator_to_raw_pointer(), Ion Gaztañaga

Boost.Iterator, boost::reverse_iterator, David Abrahams, Jeremy Siek, Thomas Witt

boost::get_pointer(), Peter Dimov & David Abrahams