Contiguous Iterators: A Refinement of Random Access Iterators

Document number: N3884
Date: 2014-01-20
Project: Programming Language C++, Library Evolution Working Group
Reply-to: Nevin "☺" Liber mailto:nliber@drw.com

Introduction

This is a proposal to add contiguous iterators to the standard, which is a refinement of random access iterators.

A contiguous iterator is a random access iterator that also meets the following requirements:

std::pointer_from(i) == std::addressof(*i) (when i is dereferenceable)

std::pointer_from(i + n) == std::pointer_from(i) + n (when i + n is a valid iterator)

Motivation and Scope

It is very convenient to know when a given range [i1..i2) has contiguous memory. Some examples:

- Passing buffers to C APIs
- Algorithm improvements; e.g., memcpy a contiguous POD
- (Proposed) string_view can be constructed from any contiguous range of chars

Currently, we just "know" (based on reading the standard) that array, basic_string, vector have contiguous storage, but we have no standard way to programmatically determine it other than by hardcoding the types, which is not a scalable solution.

As the examples show, there is a need to get to the underlying storage from just the iterator itself. We've seen developers use the sometimes undefined "&*i" construct (undefined when i "points" just past the end of a buffer). We can do better.

Impact on the Standard

The following is a summary of what needs to be changed in the standard to support contiguous iterators.

- A definition and requirements for contiguous iterators shall be added to 24.2 [iterator.requirements].
- A definition for std::pointer_from and std::do_pointer_from shall be added to <iterator>
- A contiguous_iterator_tag publicly derived from random_access_iterator_tag shall be added to <iterator>.
- iterator_traits which are specialized on pointers and pointers to const will have their iterator_category changed from random_access_iterator_tag to contiguous_iterator_tag.
- The standard library types basic_string, array, vector and valarray shall change their iterator types from random access iterators to contiguous iterators.
- move_iterator<ContiguousIterator> shall have added support for conversion to T* via do_pointer_from.

Design Decisions

contiguous_iterator_tag:

There needs to be a way to programmatically determine if an iterator "points" to contiguous space, and that method is done with tags, so I am proposing that we add a contiguous_iterator_tag.

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.

std::pointer_from:

As I alluded to earlier, it is desirable to get to the underlying contiguous storage from just the iterators without requiring the container as well. To discourage unsafe ways of doing so, I shall provide such a method.

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

The initial thought was to have an explicit cast to a T* can cover both pointers and non-pointer contiguous iterators, but some people consider that going too far towards weakening the boundary between pointers and iterators.

Given that concern, 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. std::do_pointer_from will have implementations for pointers (which just return the pointer) and the following standard library types: array, basic_string, move_iterator, vector and valarray. This gives container implementers the most flexibility. Also, such a mechanism could be extended to support other conversions to raw pointers, such as from shared_ptr, unique_ptr or other smart pointers.

Can do_pointer_from can return a custom class that is implicitly convertible to T*. I'd say no. It complicates usage such as "auto p = std::pointer_from(i);", which itself could be addressed with, say, a traits class derived from iterator_traits (I don't want to use iterator_traits itself because this mechanism can be generalized to more than just iterators), but that is a lot of machinery for something that doesn't have a use case. do_pointer_from is already a customization point; making it support weird cases just makes all uses more complicated, as well as the name itself would be misleading.

Some people are unsure whether move iterators should be considered to be contiguous, because turning a move_iterator into a raw pointer loses its move semantics.

Other thoughts:

I believe this is the first library feature which uses argument dependent lookup (range-based for does, but that is a core language feature) and LEWG should affirm this direction for this and future libraries, especially since it comes close to reserving a function name in the user's own namespace. For instance: should there be a naming convention for these kinds of functions?

A number of people have mentioned that the current iterator categories are broken because they combine traversal properties with dereference properties, and would prefer a redesign at some point. However, this proposal doesn't make things any worse in that regard.

Note: The names contiguous_iterator_tag, pointer_from and do_pointer_from are placeholders for actual names to be determined by the committee.

Techincal Specifications
The following are the proposed changes applied to n3797:

21.4p3 [basic.string]
The iterators supported by basic_string are ~~random access~~ contiguous iterators (24.2.8).

21.3
//21.9 iterator support
template<class charT, class traits, class Allocator>
charT* do_pointer_from(typename basic_string<charT, traits, Allocator>::iterator i);
template<class charT, class traits, class Allocator>
const charT* do_pointer_from(typename basic_string<charT, traits, Allocator>::const_iterator i);

21.9 [basic.string.iterator]
The template functions do_pointer_from shall meet the contiguous iterator requirements (24.2.8) and pointer access (24.8).

23.2.1p9 [container.requirements.general]
If the iterator type of a container belongs to the contiguous iterator category, the container is called contiguous and satisfies the additional requirements in Table 97 ½.

Table 97 ½ Contiguous Container Requirements
Expression: std::do_pointer_from(iterator i)
Return type: T*
Complexity: constant

Expression: std::do_pointer_from(const_iterator i)
Return type: const T*
Complexity: constant

23.3.1p2 (in the appropriate places) [sequences.general]
Header <array> synopsis
…
template<class T, size_t N>
T* do_pointer_from(typename array<T, N>::iterator i);
template<class T, size_t N>
const T* do_pointer_from(typename array<T, N>::const_iterator i);
…

Header <vector> synopsis
…
template<class T, class Allocator>
T* do_pointer_from(typename vector<T, A>::iterator i);
template<class T, class Allocator>
const T* do_pointer_from(typename vector<T, A>::const_iterator i);
…

23.3.2.1 [array.overview]
The header <array> defines a class template for storing fixed-size sequences of objects. An array supports ~~random access~~ contiguous iterators. An instance of array<T, N> stores N elements of type T , so that size() == N is an invariant. The elements of an array are stored contiguously, meaning that if a is an array<T, N> then it obeys the identity &a[n] == &a[0] + n for all 0 <= n < N .

23.3.6.1 [vector.overview]
A vector is a sequence container that supports ~~random access~~ contiguous iterators. In addition, it supports (amortized) constant time insert and erase operations at the end; insert and erase in the middle take linear

time. Storage management is handled automatically, though hints can be given to improve efficiency. The elements of a vector are stored contiguously, meaning that if v is a vector<T, Allocator> where T is some type other than bool , then it obeys the identity &v[n] == &v[0] + n for all 0 <= n < v.size() .

23.3.6.2p10 [vector.cons]
Complexity: Makes only N calls to the copy constructor of T (where N is the distance between first and last) and no reallocations if iterators first and last are of forward, bidirectional, ~~or~~ random access or contiguous categories. It makes order N calls to the copy constructor of T and order log(N) reallocations if they are just input iterators.

23.6.4p1 [priority.queue]
Any sequence container with random access or contiguous iterator and supporting operations front() , push_back() and pop_back() can be used to instantiate priority_queue . In particular, vector (23.3.6) and deque (23.3.3) can be used. Instantiating priority_queue also involves supplying a function or function object for making priority comparisons; the library assumes that the function or function object defines a strict weak ordering (25.4).

24.2.1p2 [iterator.requirements.general]
This International Standard defines ~~five~~ six categories of iterators, according to the operations defined on them: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators*, ~~and~~ *random access iterators* ~~,~~ and *contiguous iterators* as shown in Table 105 .

Table 105 — Relations among iterator categories

```
Contiguous -> Random Access -> Bidirectional -> Forward -> Input
                                                        -> Output
```

24.2.1p3 [iterator.requirements.general]
Forward iterators satisfy all the requirements of input iterators and can be used whenever an input iterator is specified; Bidirectional iterators also satisfy all the requirements of forward iterators and can be used whenever a forward iterator is specified; Random access iterators also satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified; Contiguous iterators also satisfy all the requirements of random access iterators and can be used whenever a random access iterator is specified.

24.2.8 [contiguous.iterators]
A class or pointer type X satisfies the requirements of a contiguous iterator if, in addition to satisfying the requirements for random access iterators, the following expressions are valid as shown in Table 111 ½.

Table 111 ½ – Contiguous iterator requirements (in addition to random access iterator)

Expression: std::pointer_from(a)
Return type: reference
Operational semantics: if a is dereferenceable, std::address_of(*a); otherwise, if a is reachable from a dereferenceable iterator b, std::pointer_from(b) + (a – b); otherwise a valid pointer value ([basic.compound]).

Expression:
    std::pointer_from(a + n)
    std::pointer_from(n + a)
Return type: T*
Operational semantics: std::pointer_from(a) + n

24.3 [iterator.synopsis]
…
struct random_access_iterator_tag: public bidirectional_iterator_tag { };

struct contiguous_iterator_tag: public random_access_iterator_tag { };
…
// 24.5, predefined iterators
…
template<class Iterator> move_iterator;
…
template<class Iterator>
auto do_pointer_from(move_iterator<Iterator> i) -> decltype(pointer_from(i.base()));
…
// 24.8, pointer access
template<class I> auto pointer_from(I i) -> decltype(do_pointer_from(i));
template<class T> T* do_pointer_from(T* p);
template<class T> const T* do_pointer_from(const T* p);

24.4.1p3 [iterator.traits]
It is specialized for pointers as
namespace std {
template<class T> struct iterator_traits<T*> {
typedef ptrdiff_t difference_type;
typedef T value_type;
typedef T* pointer;
typedef T& reference;
typedef ~~random_access_iterator_tag~~ contiguous_iterator_tag iterator_category;
};
}
and for pointers to const as
namespace std {
template<class T> struct iterator_traits<const T*> {
typedef ptrdiff_t difference_type;
typedef T value_type;
typedef const T* pointer;
typedef const T& reference;
typedef ~~random_access_iterator_tag~~ contiguous_iterator_tag iterator_category;
};
}

24.4.1p4 [iterator.traits]
[Note: If there is an additional pointer type _ _ far such that the difference of two _ _ far is of type long , an implementation may define
template<class T> struct iterator_traits<T _ _ far*> {
typedef long difference_type;
typedef T value_type;
typedef T _ _ far* pointer;
typedef T _ _ far& reference;
typedef ~~random_access_iterator_tag~~ contiguous_iterator_tag iterator_category;
};
—end note]

24.4.3p1 [std.iterator.tags]
It is often desirable for a function template specialization to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces category tag classes which are used as compile time tags for algorithm selection. They are: input_iterator_tag , output_iterator_tag , forward_iterator_tag , bidirectional_iterator_tag ~~and~~, random_access_iterator_tag and contiguous_iterator_tag . For every iterator of type Iterator , iterator_traits<Iterator>::iterator_category shall be defined to be the most specific category tag that describes the iterator's behavior.

namespace std {
struct input_iterator_tag { };

struct output_iterator_tag { };
struct forward_iterator_tag: public input_iterator_tag { };
struct bidirectional_iterator_tag: public forward_iterator_tag { };
struct random_access_iterator_tag: public bidirectional_iterator_tag { };
struct contiguous_iterator_tag: public random_access_iterator_tag { };
}

24.4.4p1 [iterator.operations]
Since only random access and contiguous iterators provide + and - operators, the library provides two function templates advance and distance . These function templates use + and - for random access and contiguous iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use ++ to provide linear time implementations.

24.4.4p2 [iterator.operations]
Requires: n shall be negative only for bidirectional ~~and~~, random access and contiguous iterators.

24.4.4p4 [iterator.operations]
Effects: If InputIterator meets the requirements of random access or contiguous iterator, returns (last - first); otherwise, returns the number of increments needed to get from first to last.

24.4.4p5 [iterator.operations]
Requires: If InputIterator meets the requirements of random access or contiguous iterator, last shall be reachable from first or first shall be reachable from last; otherwise, last shall be reachable from first.

24.5.3.2p1 [move.iter.requirements]
The template parameter Iterator shall meet the requirements for an Input Iterator (24.2.3). Additionally, if any of the bidirectional ~~or~~, random access or contiguous traversal or conversion functions are instantiated, the template parameter shall meet the requirements for a Bidirectional Iterator (24.2.6) ~~or~~, a Random Access Iterator (24.2.7)~~,~~ or a Contiguous Iterator (24.2.8) respectively.

24.5.3.3.x move_iterator conversion to pointer [move.iter.conversion.ptr]
template<class Iterator>
auto do_pointer_from(move_iterator<Iterator> i) -> decltype(pointer_from(i.base()));

24.8 pointer access
template<class I>
auto pointer_from(I i) -> decltype(do_pointer_from(i));

Returns: The result of an ordinary unqualified lookup (3.4.1) call to do_pointer_from(i)
Complexity: constant

[Note: The intent is that a program can define a constant time do_pointer_from function in the same namespace as a custom type for converting that custom type to a raw pointer, while generic code should call std::pointer_from to perform the conversion. An example would be a contiguous iterator for a custom container. – end note]

template<class T>
T* do_pointer_from(T* p);

Returns: p
Complexity: constant

Template<class T>
const T* do_pointer_from(const T* p);

Returns: p
Complexity: constant

25.1p5 [algorithms.general]
Throughout this Clause, the names of template parameters are used to express type requirements. If an algorithm's template parameter is InputIterator , InputIterator1 , or InputIterator2 , the actual template argument shall satisfy the requirements of an input iterator (24.2.3). If an algorithm's template parameter is OutputIterator , OutputIterator1 , or OutputIterator2 , the actual template argument shall satisfy the requirements of an output iterator (24.2.4). If an algorithm's template parameter is ForwardIterator , ForwardIterator1 , or ForwardIterator2 , the actual template argument shall satisfy the requirements of a forward iterator (24.2.5). If an algorithm's template parameter is BidirectionalIterator , Bidirectional-Iterator1 , or BidirectionalIterator2 , the actual template argument shall satisfy the requirements of a bidirectional iterator (24.2.6). If an algorithm's template parameter is RandomAccessIterator , Random-AccessIterator1 , or RandomAccessIterator2 , the actual template argument shall satisfy the requirements of a random-access iterator (24.2.7). If an algorithm's template parameter is ContiguousIterator, ContiguousIterator1 or ContiguousIterator2, the actual template argument shall satisfy the requirements of a contiguous iterator (24.2.8).

25.2.11p3 [alg.equal]
Complexity: No applications of the corresponding predicate if InputIterator1 and InputIterator2 meet the requirements of random access or contiguous iterators and last1 - first1 != last2 - first2. Otherwise, at most min(last1 - first1, last2 - first2) applications of the corresponding predicate.

25.2.12p4 [alg.is_permutation]
Complexity: No applications of the corresponding predicate if ForwardIterator1 and ForwardIterator2 meet the requirements of random access or contiguous iterators and last1 - first1 != last2 - first2. Otherwise, exactly distance(first1, last1) applications of the corresponding predicate if equal(first1, last1, first2, last2) would return true if pred was not given in the argument list or equal(first1, last1, first2, last2, pred) would return true if pred was given in the argument list; otherwise, at worst $O(N^2)$, where N has the value distance(first1, last1).

25.4.3p1 [alg.binary.search]
All of the algorithms in this section are versions of binary search and assume that the sequence being searched is partitioned with respect to an expression formed by binding the search key to an argument of the implied or explicit comparison function. They work on non-random access and non-contiguous iterators minimizing the number of comparisons, which will be logarithmic for all types of iterators. They are especially appropriate for random access and contiguous iterators, because these algorithms do a logarithmic number of steps through the data structure. For non-random access and non-contiguous iterators they execute a linear number of steps.

25.4.6p1 [heap.operations]
A heap is a particular organization of elements in a range between two random access or contiguous iterators [a,b) . Its two key properties are: (1) There is no element greater than *a in the range and (2) *a may be removed by pop_heap() , or a new element added by push_heap() , in O(log(N)) time.

26.5.1.2 [rand.req.seedseq]
A class S satisfies the requirements of a seed sequence if the expressions shown in Table 115 are valid and have the indicated semantics, and if S also satisfies all other requirements of this section 26.5.1.2 . In that Table and throughout this section:
a) T is the type named by S 's associated result_type ;
b) q is a value of S and r is a possibly const value of S ;
c) ib and ie are input iterators with an unsigned integer value_type of at least 32 bits;
d) rb and re are mutable random access or contiguous iterators with an unsigned integer value_type of at least 32
bits;
e) ob is an output iterator; and
f) il is a value of initializer_list<T> .

26.6.1 [valarray.syn]
Header <valarray> synopsis
…

```
template <class T> unspecified 1 begin(valarray<T>& v);
template <class T> unspecified 2 begin(const valarray<T>& v);
template <class T> unspecified 1 end(valarray<T>& v);
template <class T> unspecified 2 end(const valarray<T>& v);

template<class T> T* do_pointer_from(unspecified1);
template<class T> const T* do_pointer_from(unspecified2);
```

26.6.10.1 [valarray.range]
In the begin and end function templates that follow, unspecified 1 is a type that meets the requirements of a mutable ~~random access~~ contiguous iterator (24.2.7) whose value_type is the template parameter T and whose reference type is T& . unspecified 2 is a type that meets the requirements of a constant ~~random access contiguous~~ iterator (24.2.7) whose value_type is the template parameter T and whose reference type is const T& . The do_pointer_from function templates will in constant time convert unspecified1 to T* and unspecifed2 to const T* respectively.

C.? C++ and ISO C++1y
C.?.24 Code that specifically looks for a random_access_iterator_tag (such as in a template specialization) may break or perform sub-optimally.

C.?.24 Code which defines the function do_pointer_from(T) in their own namespace may cause breakage.

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 Marc Glisse for pointing out that the pointer conversion should point to the same object.

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

Special thanks to Tom Rodgers and Anthony Syzdek for proofreading it back at the office.

Special thanks to 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 and Mathias Gaunard for helping to flush this out on the LEWG mailing list.

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

References

string_view: a non-owning reference to a string, revision 6, https://rawgithub.com/google/cxx-std-draft/string-ref-paper/string_view.html

N3797, Working Draft, Standard for Programming Language C++, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf