Project: Programming Language C++, Library Evolution Working Group
Document number: P0122R0
Date: 2015-09-25
Reply-to: Neil MacIntosh neilmac@microsoft.com

array_view: bounds-safe views for sequences of objects

Contents

Introduction

This paper presents a new design of the fundamental vocabulary type *array_view*.

The design of the *array_view* type discussed in this paper is related to the *array_view* previously proposed in N3851 [1] and also drew on ideas in the *array_ref* and *string_ref* classes proposed in N3334 [2]. It is an abstraction that provides a view over a contiguous sequence of objects the storage of which is owned by some other object.

However, it differs in key aspects outlined below. The design for *array_view* presented here provides bounds-safety guarantees through a combination of compile-time and run-time constraints.

Motivation and Scope

The evolution of the standard library has demonstrated that it is possible to design and implement abstractions in Standard C++ that improve the reliability of C++ programs without sacrificing either performance or portability. This proposal identifies a new "vocabulary type" for inclusion in the standard library that enables both high performance and bounds-safe access to contiguous sequences of elements. This type would also improve modularity, composability, and reuse by decoupling accesses to array data from the specific container types used to store that data.

These characteristics lead to higher quality programs. Some of the bounds and type safety constraints of *array_view* directly support "correct-by-construction" programming methodology – where errors simply do not compile. One of the major advantages of *array_view* over container types (or worst of all, simply a "pointer plus length" pair of parameters) is that it provides clearer semantics hints to analysis tools looking to help detect and prevent defects early in a software development cycle.

Impact on the Standard

This proposal is a pure library extension. It does not require any changes to standard classes, functions or headers. Nor does it require any changes or extensions to the core language.

However, it can be imagined that some standard library functions and classes might benefit from adopting overloads for this new type, if it is adopted.

array_view has been implemented in standard C++ and successfully used within at least one commercial codebase. An open source, reference implementation is available at https://github.com/Microsoft/GSL [3].

Design Decisions

Interface similiarities to existing array storage types

array_view's interface is largely modelled on that of *std::array*. It is intended to be able to be adopted with a minimum of source changes in code that previously used an array, a *std::array* or a pointer to access more than one object.

array_view is simply a view over another object's contiguous storage– and unlike *std::array* it does not "own" the elements that are accessible through its interface. An important observation arises from this: *array_view* never performs any free store allocations.

To emphasize this drop-in source compatibilty, *array_view* constructs or assigns readily from both built-in array objects, *std::array* and *std::vector*.

Fixed or dynamic lengths

The general usage protocol of the *array_view* class template supports both static-size (fixed at compile-time) and dynamic-size (provided at runtime) view. A fixed-size *array_view* requires no storage size overhead beyond a single pointer – using the type system to carry the fixed length information. This makes *array_view* an extremely efficient type to use for fixed-size buffer access. A dynamic-size *array_view* is, conceptually, just a pointer and size field (in one dimension, a single integer).

Template Parameters

array_view must be configured with at least one template parameter: *ValueType* – that specifies the type of the elements being accessed through the *array_view*.

The SizeType parameter can be provided and used to configure *array_view* to use a specific type for measurement of the *array_view* length and for subscript operations. This parameter defaults to *std::size_t*.

A variadic set of parameters can be used to provide the extents of each dimension of the *array_view*. The default is an extent of *dynamic_length* for a single dimension. *dynamic_length* is a unique value reserved to indicate that the length of the sequence is only known at runtime and must be stored within the *array_view*.

Value Type Semantics

array_view is designed as a value type – it is expected to be cheap to construct, copy, move and use. Users are encourage to use it as a pass-by-value parameter type wherever they would have passed a pointer or *std::array* (or a container type such as *std::vector* by-reference).

Conceptually, *array_view* is simply a pointer to some storage and a count of the elements that can be accessed via that pointer. Those two values within an *array_view* can only be set via construction or assignment (i.e. all member-functions other than constructors and assignment operators are const).

These value type characteristics also help provide compiler implementations with considerable scope for optimizing the use of *array_view* within programs.

Construction

The *array_view* class is expected to become a frequently used vocabulary type in function interfaces (as safer replacement of "(pointer, length)" idioms), as it specifies a minimal set of requirements for safely accessing a sequence of objects and decouples a function that needs to access a sequence from the details of the storage that holds such elements.

To simplify use of *array_view* as a simple parameter, *array_view* offers a number of constructors for common container types that store contiguous sequences of elements:

It is allowed to construct an *array_view* from *nullptr*, and this creates an object with *.length() == 0*. Any attempt to construct an *array_view* with a *nullptr* value and a non-zero length is considered a range-check error and causes *std::terminate()* to be called.

Element types and conversions

array_view allows specification of its element type via its first template parameter. Construction or assignment between *array_view* objects with different element types is allowed whenever it can be determined that this assignment is exactly storage-size and alignment equivalent, and that the types allow legal conversion. So, for example, conversion between *array_view<int>* and *array_view<unsigned int>* is allowed.

It is always possible, as a result of these rules, to convert from an *array_view<T>* to an *array_view<const T>*. It is not allowed to convert in the opposite direction, from *array_view<const T>* to *array_view<T>*. This property is extremely convenient for calling functions.

Byte representations and conversions

array_view depends upon a distinct "byte" type that represents a single addressable byte on any system, for object representation – in preference to common practice of using character types for this purpose. Such a type could be defined in the standard library as simply as:

```
enum class byte : std::uint8_t {};
```

The critical features of this type are:

- it improves type-safety by distinguishing between byte-oriented access to memory and accessing memory as a character or integral value
- it does not support arithmetic operations on *byte* values (they are storage, not arithmetic types)
- it makes the intent of code clearer to the reader (and tooling)

An *array_view* of any element type that is a standard layout type can be converted to an *array_view<const byte>* or an *array_view<byte>* via the member functions *as_bytes()* and *as_writeable_bytes()* respectively. These operators are considered useful for systems programming where byte-oriented access for serialization and data transmission is essential.

Conversely, an *array_view* of any standard layout type can be constructed from an *array_view* of *byte*, as long as there are sufficient *byte*s to convert to a whole number of elements of the target type.

These byte-representation conversions still preserve const-correctness, however. It is not possible to convert from an *array_view* of const bytes to an array of non-const T. Nor can an *array_view* of const T be converted to an *array_view* of non-const bytes.

Alignment considerations are disregarded in these byte-representation conversions. It is left to the user to ensure any alignment requirements are correctly considered before performing conversions.

Range-checking and bounds-safety

All accesses to the data encapsulated by an *array_view* are range-checked to ensure they remain within the bounds of the *array_view*. Any failure to meet *array_view*'s bounds-safety constraints will result in a call to *std::terminate()* (a "fail-fast" approach to safety). This is a critical aspect of *array_view*'s design, and

allows users to rely on the guarantee that as long as a sequence is accessed via a correctly initialized *array_view*, then its bounds cannot be overrun.

Conversions between fixed-size and dynamic-size *array_view* objects are allowed, but with strict constraints to ensure bounds-safety is always preserved. In each case, the following rules assume the element types of the *array_view* objects are compatible for assignment as described earlier.

1. Any fixed-size *array_view* may be constructed or assigned from another fixed-size array_view of equal or greater length.
2. A dynamic-size *array_view* may always be constructed or assigned from a fixed-size *array_view*.
3. A fixed-size *array_view* may always be constructed or assigned from a dynamic-size *array_view*. However, in this case, a runtime range-check is performed to ensure the construction or assignment is bounds-safe, and will call *std::terminate()* if it is not.

array_view supports either read-only or mutable access to the sequence it encapsulates. To access read-only data, the user can declare an *array_view<const T>*, and access to mutable data would use an *array_view<T>*.

array_view objects may have multiple dimensions, though it is envisaged that the majority of usages of this type are for single-dimension sequences of data – just as with arrays and pointers today.

strided_array_view is a related type representing a non-contiguous, regular view over a contiguous sequence of data (often, but not necessarily, an *array_view*).

Iterators

array_view provides both const and mutable random-access iterators over its data, just like *std::vector* and *std::array*. All accesses to elements made through these iterators are range-checked, just as if they had been performed via the subscript operator on *array_view*.

Comparison operations

array_view supports all the same comparison operations as a standard library container: element-wise comparison and a total ordering by lexicographical comparison. Again, this allows it to function as a drop-in replacement for parameters of existing container types like *std::array* or *std::vector*.

Regardless of whether they contain a valid pointer or *nullptr*, zero-length arrays are all considered equal. This is considered a useful property when writing library code. If users wish to distinguish between a zero-length array with a valid pointer value and an *array_view* containing *nullptr*, then they can do so by calling the *data()* member function and examining the pointer value directly.

Creating sub-views

array_*view* offers convenient member functions for generating a new *array_view* that is a reduced view over its sequence. In each case, the newly constructed *array_view* is returned by value from the member function. As the design requires bounds-safety, these member functions are guaranteed to either succeed and return a valid *array_view*, or call *std::terminate()* if the parameters were not within range.

first() returns a new *array_view* that is limited to the first N elements of the original sequence. Conversely, *last()* returns a new *array_view* that is limited to the last N elements of the original

sequence. *sub()* allows an arbitrary sub-range within the sequence to be selected and returned as a new *array_view*. All three of these members are only available on *array_view* objects of one dimension. All three are also overloaded to take their parameters as template parameters, rather than function parameters. These overloads are helpful for creating fixed-size *array_view* objects from an original input *array_view*.

Multidimensional *array_view*s may be sub-ranged using the *section()* member function, or "sliced" using the subscript operator.

Multidimensional access

As per the N3851 proposal for *array_view*, some helper types are required to simplify access to multidimensional *array_view*s [1]. The design of these types is largely unchanged from that original proposal.

When an *array_view* is used for a multidimensional view on a sequence of objects, then the underlying storage is contiguous in the least significant dimension and uniformly strided in other dimensions. *strided_array_view* is a generalization of *array_view*, where the requirement of contiguity in the least significant dimension is lifted. *strided_array_view* objects can be directly constructed, or returned from a call to *.section()* on a multidimensional *array_view*.

A *bounds* (actually *static_bounds* and *strided_bounds* concrete types) is a type that represents rectangular bounds on an N-dimensional discrete space, while *index* is a type that represents an offset or a point in such space (which can then be either mapped to a single element in an *array_view* or *strided_array_view* or considered out-of-range).

bounds_iterator is a random access iterator over an imaginary space imposed by a *bounds* object, with an *index* as its value type. It allows iterator-based algorithms to be used against a *bounds* object, which can, in turn, simplify accessing elements within a multidimensional *array_view*.

Specification

```
// class that represents a point in a multidimensional space
template <size_t Rank, typename ValueType = size_t>
class index
{
public:
      static const size_t rank = Rank;
      using value_type = ValueType;
      using reference = ValueType&;
      using const_reference = const ValueType&;

      constexpr index() noexcept;
      constexpr index(const value_type (&values)[rank]);
      constexpr index(std::initializer_list< value_type > il);
      constexpr index(const index&) = default;
```

```
      template <typename OtherValueType>
      constexpr index(const index<Rank, OtherValueType>& other);

      // Preconditions: component_idx < rank
      constexpr size_type operator[](size_type component_idx) noexcept;

      // Preconditions: component_idx < rank
      constexpr const_reference operator[](size_type component_idx) const
noexcept;

      constexpr bool operator==(const index& rhs) const noexcept;
      constexpr bool operator!=(const index& rhs) const noexcept;
      constexpr index operator+() const noexcept;
      constexpr index operator-() const noexcept;
      constexpr index operator+(const index& rhs) const noexcept;
      constexpr index operator-(const index& rhs) const noexcept;
      constexpr index& operator+=(const index& rhs) noexcept;
      constexpr index& operator-=(const index& rhs) noexcept;
      constexpr index& operator++() noexcept;
      constexpr index operator++(int) noexcept;
      constexpr index& operator--() noexcept;
      constexpr index operator--(int) noexcept;
      constexpr index operator*(value_type v) const noexcept;
      constexpr index operator/(value_type v) const noexcept;
      constexpr index& operator*=(value_type v) noexcept;
      constexpr index& operator/=(value_type v) noexcept;
      constexpr index operator*(value_type v, const index& rhs) noexcept;

private:
      // ... implementation-defined
};


// a random-access iterator over a static_bounds or strided_bounds object
// has the usual form so elided here for brevity of exposition
// comes in both const and non-const flavors
template <typename IndexType>
class bounds_iterator;

// static_bounds is a fixed set of extents
// in multidimensional space for an array_view
// this is one instance of the “bounds” conceptual type
template <typename SizeType, size_t FirstRange, size_t... RestRanges>
class static_bounds<SizeType, FirstRange, RestRanges...>
{
public:
```

```
      static const size_t rank = /* calculated from number of ranges */;
      static const SizeType static_size = /* linearized size across all
dimensions */;

      using size_type = SizeType;
      using index_type = index<rank, size_type>;
      using iterator = bounds_iterator<index_type>;
      using const_iterator = bounds_iterator<index_type>;
      using difference_type = ptrdiff_t;
      using sliced_type = static_bounds<SizeType, RestRanges...>;

      // default construction
      constexpr static_bounds() = default;

      // simple copy
      constexpr static_bounds(const static_bounds &) = default;

      // ensures OtherSizeType and Ranges are safely convertible
      template <typename OtherSizeType, size_t... Ranges>
      constexpr static_bounds(const static_bounds<OtherSizeType, Ranges...>
&other);

      constexpr static_bounds(std::initializer_list<size_type> il);

      constexpr static_bounds& operator=(const static_bounds& otherBounds);

      // returns the bounds with one dimension sliced away
      constexpr sliced_type slice() const noexcept;

      // the stride over this bounds
      constexpr size_type stride() const noexcept;

      constexpr size_type size() const noexcept;

      constexpr size_type total_size() const noexcept;

      constexpr size_type linearize(const index_type & idx) const;

      constexpr bool contains(const index_type& idx) const noexcept;

      constexpr size_type operator[](size_t index) const noexcept;

      // get the extent for a specific dimension
      template <size_t Dim = 0>
      constexpr size_type extent() const noexcept;
```

```
      // get all extents as an index value
      constexpr index_type index_bounds() const noexcept;

      template <typename OtherSizeTypes, size_t... Ranges>
      constexpr bool operator == (const static_bounds<OtherSizeTypes,
Ranges...> & rhs) const noexcept;

      template <typename OtherSizeTypes, size_t... Ranges>
      constexpr bool operator != (const static_bounds<OtherSizeTypes,
Ranges...> & rhs) const noexcept;

      constexpr const_iterator begin() const noexcept;
      constexpr const_iterator end() const noexcept;
};

// strided_bounds is an instance of the “bounds” conceptual type
// but for the bounds of a strided_array_view
// it shares the same interface as static_bounds apart from the
// following functions which are the only ones shown here for brevity
template <size_t Rank, typename SizeType = size_t>
class strided_bounds
{
public:
      static const size_t rank = /* calculated from number of ranges */;
      static const SizeType static_size = /* linearized size across all
dimensions */;

      using size_type = SizeType;
      using index_type = index<rank, size_type>;
      using iterator = bounds_iterator<index_type>;
      using const_iterator = bounds_iterator<index_type>;
      using difference_type = ptrdiff_t;
      using sliced_type = static_bounds<SizeType, RestRanges...>;

      constexpr strided_bounds(const index_type &extents, const index_type
&strides);
      constexpr strided_bounds(const value_type(&values)[rank], const
index_type& strides);
      constexpr index_type strides() const noexcept;

      // simple copy
      constexpr strided_bounds(const strided_bounds &) = default;

      // ensures OtherSizeType and Ranges are safely convertible
      template <typename OtherSizeType, size_t... Ranges>
```

```
      constexpr strided_bounds(const static_bounds<OtherSizeType, Ranges...>
&other);

      constexpr strided_bounds& operator=(const strided_bounds& otherBounds);

      // returns the bounds with one dimension sliced away
      constexpr sliced_type slice() const noexcept;

      // the stride over this bounds
      constexpr size_type stride() const noexcept;

      constexpr size_type size() const noexcept;

      constexpr size_type total_size() const noexcept;

      constexpr size_type linearize(const index_type & idx) const;

      constexpr bool contains(const index_type& idx) const noexcept;

      constexpr size_type operator[](size_t index) const noexcept;

      // get the extent for a specific dimension
      template <size_t Dim = 0>
      constexpr size_type extent() const noexcept;

      // get all extents as an index value
      constexpr index_type index_bounds() const noexcept;

      template <typename OtherSizeTypes, size_t... Ranges>
      constexpr bool operator == (const strided_bounds<OtherSizeTypes,
Ranges...> & rhs) const noexcept;

      template <typename OtherSizeTypes, size_t... Ranges>
      constexpr bool operator != (const strided_bounds<OtherSizeTypes,
Ranges...> & rhs) const noexcept;

      constexpr const_iterator begin() const noexcept;
      constexpr const_iterator end() const noexcept;
};

// a helper type that is useful to represent a dimension when
// creating and navigating strided/multidimensional arrays
template <size_t DimSize = dynamic_range>
struct dim
{
      static const size_t value = DimSize;
```

```
};

template <>
struct dim<dynamic_range>
{
      static const size_t value = dynamic_range;
      const size_t dvalue;
      dim(size_t size) : dvalue(size) {}
};


// a helper type that can be passed to the ValueTypeOpt
// parameter of array_view, in which case the size_type
// member is used to determine the type used for measurement
// and index access into the array_view.
template <typename ValueType, typename SizeType>
struct array_view_options
{
      struct array_view_traits
      {
            using value_type = ValueType;
            using size_type = SizeType;
      };
};

// a random-access iterator over an array_view or strided_array_view object
// has the usual form so elided here for brevity of exposition
// comes in both const and non-const flavors
template <typename IndexType>
class array_view_iterator;


template <typename ValueTypeOpt, size_t FirstDimension, size_t...
RestDimensions>
class array_view
{
public:
      // the following “usual” types are elided here for brevity
      using bounds_type = ... ;
      using size_type = ...;
      using pointer  =...;
      using value_type = ...;
      using index_type = ...;
      using iterator = ...;
      using const_iterator = ...;
      using reference = ...;
```

```
      using rank = ...;

      // construction from single element
      explicit constexpr array_view(reference t);
      explicit constexpr array_view(const_reference t);

      // construction from ptr, len
      constexpr array_view(pointer ptr, bounds_type bounds);

      // construction from nullptr
      constexpr array_view(std::nullptr_t);
      constexpr array_view(std::nullptr_t, size_type size);

      // default construction (results in construction with {nullptr, 0})
      constexpr array_view();

      // from n-dimensions dynamic array (e.g. new int[m][4])
      constexpr array_view(T* const& data, size_type size);

      // from n-dimensions static array
      template <typename T, size_t N>
      constexpr array_view (T (&arr)[N]);

      // from n-dimensions static array with size
      // will fail-fast if size >= N
      template <typename T, size_t N>
      constexpr array_view(T(&arr)[N], size_type size);

      // from mutable element std::array
      template <size_t N>
      constexpr array_view (std::array<std::remove_const_t<value_type>, N>&
arr);

      // from read-only element std::array
      template <size_t N>
      constexpr array_view (const std::array<std::remove_const_t<value_type>,
N>& arr);

      // from begin, end pointers.
      constexpr array_view (pointer begin, pointer end);

      // from containers that have .size() and .data() function signatures
      // (checked using SFINAE elided here for readability)
      constexpr array_view (Cont& cont);

      // simple copy
```

```
      constexpr  array_view(const array_view &) = default;

      // convertible - as long as types meet std::is_convertible<>
      // and dimensions can be calculated to be bounds-compatible
      constexpr array_view(const array_view<OtherValueTypeOpt,
OtherDimensions...> &av);

      // reshape – verifies with runtime checks that bounds are compatible
      template <typename... Dimensions2>
      constexpr array_view<ValueTypeOpt, Dimensions2::value...>
as_array_view(Dimensions2... dims);

      // to bytes array
      // only available if std::is_standard_layout<> is true
      // for value_type
      constexpr array_view<const byte> as_bytes() const noexcept;

      // only available if std::is_standard_layout<> is true
      // for value_type
      constexpr array_view<byte> as_writeable_bytes() const noexcept;

      // from bytes array
      // only available when U meets std::is_standard_layout_type<>
      // performs runtime check to ensure enough bytes for a whole
      // number of elements
      template<class U>
      constexpr array_view<U> as_array_view() const noexcept;

      // type conversion
      // ensures U meets std::is_convertible<value_type, U>
      template<class U>
      constexpr array_view<const U> as_array_view() const noexcept;

      // create sub-view from the first Count elements
      template<size_t Count>
      constexpr array_view<ValueTypeOpt, Count> first() const noexcept;

      constexpr array_view<ValueTypeOpt, dynamic_range> first(size_type
count) const noexcept;

      // create sub-view from the last Count elements
      template<size_t Count>
      constexpr array_view<ValueTypeOpt, Count> last() const noexcept;

      constexpr array_view<ValueTypeOpt, dynamic_range> last(size_type count)
const noexcept;
```

```
      // create sub-view from arbitrary sub-range
      template<size_t Offset, size_t Count>
      constexpr array_view<ValueTypeOpt, Count> sub() const noexcept;

      constexpr array_view<ValueTypeOpt, dynamic_range> sub(size_type offset,
size_type count = dynamic_range) const noexcept;

      // size
      constexpr size_type length() const noexcept;
      // size expressed in bytes
      constexpr size_type bytes() const noexcept;

      // section – arbitrary sub-ranges of multidimensional array_views
      constexpr strided_array_view<ValueTypeOpt, rank> section(index_type
origin, index_type extents) const;

      // element access for linear array_views
      constexpr reference operator[](const size_type& idx) const;

      // multidimensional element access
      constexpr array_view<ValueTypeOpt, RestDimensions...>
operator[](size_type idx) const;

      // iterators
      constexpr iterator begin() const noexcept;
      constexpr iterator end() const noexcept;

      constexpr const_iterator cbegin() const noexcept;
      constexpr const_iterator cend() const noexcept;

      constexpr reverse_iterator rbegin() const noexcept;
      constexpr reverse_iterator rend() const noexcept;

      constexpr const_reverse_iterator crbegin() const noexcept;
      constexpr const_reverse_iterator crend() const noexcept;

      // all comparison operators are only enabled
      // for compatible value and bounds type combinations
      template <typename OtherValueType, typename OtherBoundsType>
      constexpr bool operator== (const array_view<OtherValueType,
OtherBoundsType> & other) const noexcept;

      template <typename OtherValueType, typename OtherBoundsType>
      constexpr bool operator!= (const array_view<OtherValueType,
OtherBoundsType> & other) const noexcept;
```

```
      template <typename OtherValueType, typename OtherBoundsType>
      constexpr bool operator< (const array_view<OtherValueType,
OtherBoundsType> & other) const noexcept;

      template <typename OtherValueType, typename OtherBoundsType>
      constexpr bool operator<= (const array_view<OtherValueType,
OtherBoundsType> & other) const noexcept;

      template <typename OtherValueType, typename OtherBoundsType>
      constexpr bool operator> (const array_view<OtherValueType,
OtherBoundsType> & other) const noexcept;

      template <typename OtherValueType, typename OtherBoundsType>
      constexpr bool operator>= (const array_view<OtherValueType,
OtherBoundsType> & other) const noexcept;
};

// convenience free functions for performing conversions to array_view
template <typename T, size_t... Dimensions>
constexpr auto as_array_view(T* const & ptr, dim<Dimensions>... args) ->
array_view<std::remove_all_extents_t<T>, Dimensions...>;

template <typename T>
constexpr array_view<T> as_array_view (T* arr, size_t len);

template <typename T, size_t N>
constexpr array_view<T, size_t, N> as_array_view (T (&arr)[N]);

template <typename T, size_t N>
constexpr array_view<const T, N> as_array_view(const std::array<T, N> &arr);

template <typename T, size_t N>
constexpr array_view<const T, N> as_array_view(const std::array<T, N> &&) =
delete;

template <typename T, size_t N>
constexpr array_view<T, N> as_array_view(std::array<T, N> &arr);

template <typename T>
constexpr array_view<T, dynamic_range> as_array_view(T *begin, T *end);

template <typename ValueTypeOpt, size_t Rank>
class strided_array_view
{
public:
```

```
      // the following “usual” types are elided here for brevity
      using bounds_type = ... ;
      using size_type = ...;
      using pointer  =...;
      using value_type = ...;
      using index_type = ...;
      using iterator = ...;
      using const_iterator = ...;
      using reference = ...;
      using rank = ...;

      // from static array of size N
      template<size_type N>
      strided_array_view(value_type(&values)[N], bounds_type bounds);

      // from raw data
      strided_array_view(pointer ptr, size_type size, bounds_type bounds);

      // from array view
      template <size_t... Dimensions>
      strided_array_view(array_view<ValueTypeOpt, Dimensions...> av,
bounds_type bounds);

      // conversion – ensures conversion meets usual criteria
      // (std::is_convertible<ValueTypeOpt, OtherValueTypeOpt>)
      template <class OtherValueTypeOpt>
      constexpr strided_array_view(const
strided_array_view<OtherValueTypeOpt, Rank>& av);

      // convert from bytes
      // only enabled when value_type == const byte
      template <class OtherValueType>
      strided_array_view<OtherValueType> as_strided_array_view() const;

      // create an arbitrary sub-view
      strided_array_view section(index_type origin, index_type extents)
const;

      // element access
      constexpr reference operator[](const index_type& idx) const;

      // returns a slice of the multidimensional array
      // only enabled for rank > 1
      constexpr strided_array_view<value_type, rank-1> operator[](size_type
idx) const;
```

```
      // iterators
      constexpr iterator begin() const noexcept;
      constexpr iterator end() const noexcept;

      constexpr const_iterator cbegin() const noexcept;
      constexpr const_iterator cend() const noexcept;

      constexpr reverse_iterator rbegin() const noexcept;
      constexpr reverse_iterator rend() const noexcept;

      constexpr const_reverse_iterator crbegin() const noexcept;
      constexpr const_reverse_iterator crend() const noexcept;

      // all comparison operators are only enabled
      // for compatible value and bounds type combinations
      template <typename OtherValueType, typename OtherBoundsType>
      constexpr bool operator== (const strided_array_view<OtherValueType,
OtherBoundsType> & other) const noexcept;

      template <typename OtherValueType, typename OtherBoundsType>
      constexpr bool operator!= (const strided_array_view<OtherValueType,
OtherBoundsType> & other) const noexcept;

      template <typename OtherValueType, typename OtherBoundsType>
      constexpr bool operator< (const strided_array_view<OtherValueType,
OtherBoundsType> & other) const noexcept;

      template <typename OtherValueType, typename OtherBoundsType>
      constexpr bool operator<= (const strided_array_view<OtherValueType,
OtherBoundsType> & other) const noexcept;

      template <typename OtherValueType, typename OtherBoundsType>
      constexpr bool operator> (const strided_array_view<OtherValueType,
OtherBoundsType> & other) const noexcept;

      template <typename OtherValueType, typename OtherBoundsType>
      constexpr bool operator>= (const strided_array_view<OtherValueType,
OtherBoundsType> & other) const noexcept;

};
```

Acknowledgements

This work has been heavily informed by N3851 (a previous *array_view* proposal) and previous discussion amongst committee members regarding that proposal.

This version of *array_view* was designed to support the C++ Core Coding Guidelines [4] and as such, the current version reflects the input of Herb Sutter, Jim Springfield, Gabriel Dos Reis, Chris Hawblitzel, Gor Nishanov, and Dave Sielaff. Łukasz Mendakiewicz, Bjarne Stroustrup, Eric Niebler and Artur Laksberg provided helpful review of this version of *array_view* during its development.

Many thanks to Gabriel Dos Reis for review of this document during its initial development.

References

[1] Łukasz Mendakiewicz, Herb Sutter, “Multidimensional bounds, index and array_view“, 2014, WG21 Document N3851.

[2] J. Yasskin, "Proposing array_ref<T> and string_ref, N3334," 14 January 2012. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3334.html.

[3] Microsoft, “Guideline Support Library reference implementation: array_view”, https://github.com/Microsoft/GSL

[4] Bjarne Stroustrup, Herb Sutter, “C++ Core Coding Guidelines”, 2015, https://github.com/isocpp/CppCoreGuidelines