Multidimensional bounds, index and array_view

Contents
Introduction ................................................................................................................................. 2
Motivation and Scope .................................................................................................................. 2
  Parallel Programming Perspective ......................................................................................... 3
Impact on the Standard ............................................................................................................... 4
Design Decisions ....................................................................................................................... 4
  bounds and index ..................................................................................................................... 4
  array_view and strided_array_view ....................................................................................... 9
  bounds_iterator ...................................................................................................................... 16
Possible Future Extensions ........................................................................................................ 17
Prior Art ..................................................................................................................................... 18
Acknowledgments .................................................................................................................... 18
References .................................................................................................................................. 18
Appendix: Design Alternatives .................................................................................................. 19
  Names ....................................................................................................................................... 19
  bounds and index ..................................................................................................................... 19
  array_view and strided_array_view ....................................................................................... 20
Introduction

Programs performing computations on multidimensional data are relatively common (e.g. operations on dense matrices or image processing) yet there is no standardized approach in C++ to express the concept of dimensionality. This document aims to fill this gap in the Standard C++ Library by proposing the following closely related types:

- **bounds** and **index** as means of defining and addressing multidimensional discrete spaces.
- **array_view** and **strided_array_view** as multidimensional views on contiguous or strided memory ranges, respectively.
- **bounds_iterator** providing interoperability with iterator-based algorithms.

While the proposal builds on Microsoft experience of implementing and using similar **extent**, **index** and **array_view** types in their data parallel programming model – C++ AMP [1] – we believe that these concepts will also benefit a wider C++ community.

Motivation and Scope

As an example, consider a naïve matrix-vector multiplication algorithm that uses multidimensional addressing.

Assuming the operands are an $M \times N$ matrix $A$ and an $N$-vector $B$, the result is an $M \times N$ matrix $C$. Corresponding objects are defined as follows, using **vectors** for the contiguous storage.

```cpp
auto M = 32;
auto N = 64;

auto vA = vector<float>(M * N);
auto vB = vector<float>(N);
auto vC = vector<float>(M * N);
```

**array_views**, as introduced in this proposal, allow to conveniently store references to the data along with their dimensionality and size information. Note in the following snippet that for two-dimensional views $a$ and $c$ their rank must be specified explicitly (as the second template argument) as well as their extent in each dimension (as the first constructor argument), while for one-dimensional view $b$ the same information is implicit.

```cpp
auto a = array_view<float, 2>{ { M, N }, vA }; // An MxN view on vA.
auto b = array_view<float>{ vB }; // A view on vB.
auto c = array_view<float, 2>{ { M, N }, vC }; // An MxN view on vC.
```

Next, the **bounds_iterator** enables compatibility with iterator-aware algorithms, in this example allowing to iterate over the **bounds** space (equivalent to the size of the $c$ **array_view**) using the **for_each** algorithm. Dereferencing the iterator provides an **index** of each location in the space to the lambda expression, which is subsequently used to address data in the **array_views**.

```cpp
bounds_iterator<2> first = begin(c.bounds()); // Named object for clarity.
bounds_iterator<2> last = end(c.bounds());
for_each(first, last,
    [&](index<2> idx)
    // or shortly: for(auto idx : c.bounds())
    {
        float sum = 0.0f;
    }
```


It is worth noting at this point that C++ and the Standard C++ Library already allow certain patterns for expressing multidimensional data:

- **vector of vectors**, for example:
  ```cpp
global <vector<vector<float>>> A{ 32, vector<float>(64) };
```

- **array type and `std::array`**, for example:
  ```cpp
float A[32][64];
array<array<float>, 64>, 32> C;
```

Both of these approaches have drawbacks – the former does not guarantee contiguous memory allocations between the sub-vectors, which is often beneficial in performance-critical scenarios; the latter requires array extents as constant expressions¹, which inhibits flexibility. Neither allows for convenient addressing, sectioning or slicing the representation.

This proposal has two goals:

1. Provide multidimensional views over contiguous (single dimensional) storage, abstracting the allocation from the usage².
2. Enable universal multidimensional indexing, orthogonal to the former.

**Parallel Programming Perspective**

The proposed abstraction allows for a more efficient distribution of work among the processing threads by making the iteration space larger and exposing more opportunities for runtime optimizations. In this regard we have performed a simple experimental comparison of a naïve matrix multiplication algorithm (with \(O(N^3)\) complexity) using Microsoft Parallel Patterns Library [2] and the following two approaches (also see Figure 1):

- “p_f + p_f” – parallelizing the two outermost for loops.
- “MD p_f e” – collapsing the two outermost for loops into a single loop over two-dimensional bounds and parallelizing the same.

---

¹ This concern is partially addressed with `dynarray` and arrays of runtime bound introduced in the Array Extensions TS [1].
² Parallels can be drawn in this regard with the `string_view` proposal [7].
The obtained results indicate the advantage of the second approach, enabled by the proposal. For 1500x1500 matrix the improvement of the multidimensional indexing is close to 5%, with less than 1.5% RSD. Arguably, the same effect should be achieved if the user had chosen to manually flatten, parallelize and then restructure the iteration space, it would however require sacrificing the logical structure of the code and be error-prone.

Impact on the Standard
These changes are entirely based on library extensions and do not require any language features nor changes in existing libraries.

Design Decisions

bounds 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 in practice e.g. maps to a single element in an array_view).

array_view is a multidimensional view on a storage contiguous in the least significant dimension and uniformly strided in other dimensions. The type provides member functions for accessing the underlying elements and reshaping the view. strided_array_view is a generalization of array_view, where the requirement of contiguity in the least significant dimension is lifted.

bounds_iterator is a constant random access iterator over an imaginary space imposed by a bounds object, with an index as its value type. It provides interoperability of the multidimensional structures with the traditional iterator-based algorithms.

The following sections will describe the above types in greater detail.

bounds and index
We will be discussing these two types together as they share many characteristics. On a high level, the index can be regarded as an N-dimensional vector (as a geometric quantity), while the bounds as an N-
dimensional axis-aligned rectangle\(^3\) with the minimum point at 0. (See also: Appendix: Design Alternatives, I and II).

![Graphical representation in the two-dimensional case.](image)

The type of each component in both the index and bounds is `ptrdiff_t\(^4\)`. The rationale behind is to be able for the index to both address every byte in the largest allocation and express any offset in the same. (See also: Appendix: Design Alternatives, III and IV).

There are additional invariants imposed on bounds, to which users must adhere when modifying the object:

1) every component must be greater than or equal to zero
2) product of all components must not overflow `ptrdiff_t`

Although the invariants may appear overly strict, we believe they should be trivially satisfiable in practice.

**Definitions**

bounds and index are template classes with a single template parameter designating their rank (i.e. the number of represented dimensions).

```cpp
template <int Rank> class bounds;
template <int Rank> class index;
```

Rank is required to be greater than zero\(^5\).

Both classes define the same set of nested types used later in their interfaces and expose their rank value:

```cpp
static constexpr int rank = Rank;
using reference = ptrdiff_t&;
using const_reference = const ptrdiff_t&;
using size_type = size_t;
using value_type = ptrdiff_t;
```

\(^3\) For most cases it can be thought of as a maximum point of such rectangle.

\(^4\) Should the LWG issue 2251 ("C++ library should define ssize_t") be resolved, ssize_t might have been a better choice.

\(^5\) Despite this fact, the template parameter is typed int for more robust error detection – passing a negative value can be diagnosed in a static assertion.
**Construction and Assignment**

Analogous sets of constructors and assignment operators are available for `bounds` and `index`.

```cpp
// For bounds<rank>:
constexpr bounds() noexcept;
constexpr bounds(value_type) noexcept;
constexpr bounds(const initializer_list<value_type>&) noexcept;
constexpr bounds(const bounds&) noexcept;
bounds& operator=(const bounds&) noexcept;

// For index<rank>:
constexpr index() noexcept;
constexpr index(value_type) noexcept;
constexpr index(const initializer_list<value_type>&) noexcept;
constexpr index(const index&) noexcept;
index& operator=(const index&) noexcept;
```

The default constructor zero-initializes all components.

There exists an implicit constructor from `value_type`, allowed only for rank = 1, which sets the single component to the provided value. This allows the programmer to write `cont[0]` rather than `cont[{0}]` when indexing one-dimensional containers. Analogous constructor is provided for `bounds` for commonality. (See also: Appendix: Design Alternatives, V).

The `initializer_list` constructor is assigning the initializer list elements to the `bounds` or the `index` components. The size of the `initializer_list` must be equal to the rank of the `bounds/index` type. (See also: Appendix: Design Alternatives, VI).

Copy constructors and copy assignment operators for both types copy or assign all components element-wise.

**Accessing Components**

The `bounds` and the `index` components can be accessed with the same set of subscript operator overloads.

```cpp
reference operator[](size_type component_idx) noexcept;
constexpr const_reference operator[](size_type component_idx) const noexcept;
```

The functions return references to the component at the requested zero-based index. The precondition that the argument must be less than rank applies to both functions.

**Comparison Operators**

Comparison is supported between two `bounds` objects of the same rank or two `index` objects of the same rank (but not between `bounds` and an `index`).

```cpp
// For bounds<rank>:
bool operator==(const bounds& rhs) const noexcept;
bool operator!=(const bounds& rhs) const noexcept;

// For index<rank>:
bool operator==(const index& rhs) const noexcept;
bool operator!=(const index& rhs) const noexcept;
```

`operator==` returns true iff all corresponding components of the two operands are equal. `operator!=` returns true iff there is at least one pair of corresponding components that is not equal.
Arithmetic Operators

The arithmetic operators available for the *bounds* and the *index* are different following the differences in their semantics. Generally, operations for these types follow the N-dimensional rectangle and the N-dimensional vector models, accordingly – see Table 1 for an overview.

No overflow checking is performed on any of the following operations.

*Table 1 Arithmetic operations allowed for bounds and an index; using notation: operand 1 type ⊗ operand 2 type → result type, with the permitted operators listed below.*

<table>
<thead>
<tr>
<th>bounds&lt;N&gt;</th>
<th>index&lt;N&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounds&lt;N&gt; ⊗ index&lt;N&gt; → bounds&lt;N&gt;</td>
<td>index&lt;N&gt; ⊗ index&lt;N&gt; → index&lt;N&gt;</td>
</tr>
<tr>
<td>+ - += -=</td>
<td>+ += -=</td>
</tr>
<tr>
<td>index&lt;N&gt; ⊗ bounds&lt;N&gt; → bounds&lt;N&gt;</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
</tr>
<tr>
<td>bounds&lt;N&gt; ⊗ arithmetic type → bounds&lt;N&gt;</td>
<td>index&lt;N&gt; ⊗ arithmetic type → index&lt;N&gt;</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>*= /=</td>
<td>*= /=</td>
</tr>
<tr>
<td>arithmetic type ⊗ bounds&lt;N&gt; → bounds&lt;N&gt;</td>
<td>arithmetic type ⊗ index&lt;N&gt; → index&lt;N&gt;</td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>⊗ index&lt;N&gt; → index&lt;N&gt;</td>
</tr>
<tr>
<td>+ -</td>
<td>+ -</td>
</tr>
<tr>
<td>++ --</td>
<td>(for N = 1, and also post-variants)</td>
</tr>
</tbody>
</table>

Arithmetic Operators for *bounds*

*bounds* support a range of addition and subtraction operators with an *index* of the same rank as the other operand.

```
// Members of bounds<rank>:
bounds operator+(const index<rank>& rhs) const noexcept;
bounds operator-(const index<rank>& rhs) const noexcept;
bounds& operator+=(const index<rank>& rhs) noexcept;
bounds& operator-=(const index<rank>& rhs) noexcept;

// In the namespace scope:
bounds operator+(const index<rank>&lhs, const bounds& rhs) noexcept;
```

The member binary *operator*+ and *operator*- perform the corresponding operation component-wise on a copy of *this* and the function argument, returning the copy. *operator*+= and *operator*-= work analogously, operating on and returning *this*.

For example:

```
auto bnd1 = bounds<3>{ 3, 1, 4 };
auto idx = index<3>{ 2, -1, 0 };

tests<3> bnd2 = bnd1 + idx; // bnd2 is { 5, 0, 4 }
bnd1 -= idx; // bnd1 is { 1, 2, 4 }
```
Arithmetic Operators for **index**

**index** supports a range of addition and subtraction operators between objects of the same rank.

```cpp
index  operator+(const index& rhs) const noexcept;
index  operator-(const index& rhs) const noexcept;
index& operator+=(const index& rhs) noexcept;
index& operator-=(const index& rhs) noexcept;
```

The binary **operator+** and **operator-** perform the corresponding operation component-wise on a copy of *this and the function argument, returning the copy. **operator+=** and **operator-=** work analogously, operating on and returning *this.*

```cpp
index& operator++() noexcept;
index  operator++(int) noexcept;
index& operator--() noexcept;
index  operator--(int) noexcept;
```

Pre- and post-increment and decrement operators have the traditional semantics, and as such are allowed only on **index** with rank = 1. This decision follows the logic that e.g. `++idx` shall be equivalent to `idx += 1`, which with implicit constructor is naturally supported for `rank = 1` as `idx += index<1>{ 1 }`, while ill-formed for any other rank. (See also: Appendix: Design Alternatives, VII).

```cpp
index  operator+() const noexcept;
index  operator-() const noexcept;
```

The unary **operator+** returns *this, while the unary **operator-** returns a copy of the object with all components negated.

Scaling Operators for **bounds** and **index**
The **bounds** and **index** types support an analogous set of scaling operators.

```cpp
// Members of bounds<rank>:
template <typename ArithmeticType> bounds operator*(ArithmeticType v) const noexcept;
template <typename ArithmeticType> bounds operator/(ArithmeticType v) const noexcept;
template <typename ArithmeticType> bounds& operator*=(ArithmeticType v) noexcept;
template <typename ArithmeticType> bounds& operator/=(ArithmeticType v) noexcept;

// Members of index<rank>:
template <typename ArithmeticType> index  operator*(ArithmeticType v) const noexcept;
template <typename ArithmeticType> index  operator/(ArithmeticType v) const noexcept;
template <typename ArithmeticType> index& operator*=(ArithmeticType v) noexcept;
template <typename ArithmeticType> index& operator/=(ArithmeticType v) noexcept;

// In the namespace scope:
bounds operator*(ArithmeticType v, const bounds& rhs) noexcept;
index   operator*(ArithmeticType v, const index& rhs) noexcept;
```

Any of the above functions shall not participate in overload resolution unless is_arithmetic<ArithmeticType>::value is true.

The operators are defined as template functions and accept any arithmetic type, performing the corresponding operation on each component of the **bounds** or the **index** object with the value of the deduced type, following the usual arithmetic conversions, before being implicitly converted to the **bounds** or the **index value_type**. In pseudo code:
\[ \text{result} \[i\] = \text{lhs} \[i\] \odot v, \text{ for } i=0..\text{rank}-1 \]

For example:

\[
\begin{align*}
\text{index<2> idx} &\{ 2, 3 \}; \\
\text{index<2> res} &= \text{idx} \times 1.5; \quad // \text{res is \{3, 4\}}
\end{align*}
\]

**bounds Functions**

The **bounds** type defines the following member functions.

- **constexpr size_type** size() const noexcept;
- **bool** contains(const index<rank>& idx) const noexcept;
- **bounds_iterator<rank>** begin() const noexcept;
- **bounds_iterator<rank>** end() const noexcept;

**size()** returns a hyper volume of the rectangular space enclosed by *this, i.e. the product of all components. With the aforementioned preconditions on all operations on **bounds**, the result will be always well-formed.

**contains()** checks whether the passed **index** is contained within **bounds** – returns true iff every component of idx is equal or greater than zero and is less than the corresponding component of *this.

**begin()** and **end()** return **bounds_iterator** for the space defined by *this and will be further discussed in the **bounds_iterator** section.

**array_view and strided_array_view**

The **array_view** and the **strided_array_view** represent multidimensional views onto regular collections of objects of a uniform type. The view semantics convey that objects of these types do not store the actual data, but instead enables patterns congruent to that of random access iterators or pointers. This enables the primary feature of **array_view** and **strided_array_view** – the capability to lift an arbitrary regular collection into a logical multidimensional representation.

The collection over which the view will be created must be provided as a pointer or a reference to an array or to a container. The size of the collection can be implied in limited cases, however usually it will be explicitly specified by the user. For details, refer to the **array_view Construction** section.

The requirement on the shape of the furnished collection is the primary difference between the two types. The **array_view** requires the data to be contiguous with a constant stride for each dimension, equal to 1 for the least significant dimension and increasing by the factor of the one less significant dimension’s size for each more significant dimension. Colloquially, the **array_view** stride is identical with the stride of a multidimensional array type (e.g. int arr[3][4][5]). The **strided_array_view** requires only a constant stride for each dimension (most notably, the requirement of the unitary stride in the least significant dimension is relaxed). It is used primarily to express sections of an **array_view**, however more advanced use cases are possible – e.g. defining a view over specific subobjects in a collection of POD objects. An **array_view** is implicitly convertible to a corresponding **strided_array_view**. (See also: Appendix: Design Alternatives, VIII and IX).

The view semantics (or the “pointer semantics”) of the **array_view** and the **strided_array_view** lead to a principle that any operation that invalidates a pointer in the range over which the view is created (i.e. [av.data(), av.data() + av.size()]) for **array_view**, or its generalization for **strided_array_view** invalidates
pointers and references returned from the view’s methods. A corresponding rule of thumb is that the underlying container must remain in place as long as the view is used.

The corollary is a guarantee of coherence between the view and the underlying data, as if the data was accessed through a pointer indirection.

For example:

```cpp
auto vec = vector<int>(10);
auto view = array_view<int>{ vec };
view[0] = 42;
int v = vec[0]; // v == 42
```

Applying cv-qualifiers to Views and Their Element Types

A view can be created over an arbitrary value_type, as long as there exists a conversion from the pointer to the underlying collection object type to the pointer to the value_type. This allows to distinguish two levels of constness, analogously to the pointer semantics – the constness of the view and the constness of the data over which the view is defined – see Table 2. Interfaces of both the array_view and the strided_array_view allow for implicit conversions from non-const-qualified views to const-qualified views.

Table 2 Examples of the array_view constness duality (note strided_array_view is analogous).

<table>
<thead>
<tr>
<th>...over...</th>
<th>Mutable view... (view = another_view is allowed)</th>
<th>Constant view... (view = another_view is disallowed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...mutable data (view[1] = 42 is allowed)</td>
<td>array_view&lt;int&gt; view c.f. int* view</td>
<td>const array_view&lt;int&gt; view c.f. int* const view</td>
</tr>
<tr>
<td>...constant data (view[1] = 42 is disallowed)</td>
<td>array_view&lt;const int&gt; view c.f. const int* view</td>
<td>const array_view&lt;const int&gt; view c.f. const int* const view</td>
</tr>
</tbody>
</table>

Relations Between the Types

The summary of relations between the proposed types and the select predating concepts are displayed in Figure 3. Detailed semantics of the operations are described in the following sections.

---

6 This rule is aligned with the string_view proposal [7].
Furthermore, the following implicit conversions are allowed between the view types, considering the const qualifier alone (the volatile qualifier is treated analogously):

- `array_view<T, N> → array_view<const T, N>`
- `array_view<T, N> → strided_array_view<T, N>`
- `array_view<T, N> → strided_array_view<const T, N>`
- `strided_array_view<T, N> → strided_array_view<const T, N>`

Definitions
The `array_view` and the `strided_array_view` are template classes with two template parameters – a type template parameter designating the type the view presents; and an integral template parameter designating the rank of the view.

```
template <typename ValueType, int Rank = 1> class array_view;
template <typename ValueType, int Rank = 1> class strided_array_view;
```

Rank is defined as `int` for commonality with `bounds` and `index` types; and similarly is required to be greater than zero. Its default value is 1 for convenience.

The `array_view` and the `strided_array_view` define the same set of nested types used later in their interfaces and expose their rank value:

```
static constexpr int rank = Rank;
using index_type = index<rank>;
using bounds_type = bounds<rank>;
using size_type = typename bounds_type::size_type;
using value_type = ValueType;
using pointer = typename add_pointer_t<value_type>;
using reference = typename add_lvalue_reference_t<value_type>;
```

`array_view` Construction and Assignment
Some of the following functions refer to a `Container` concept, which for the purpose of this document is defined as a type which:

1) `size()` member function returns a type convertible to `bounds<rank>::value_type`
2) `data()` member function returns a type convertible to `value_type*` designating the address of the first of `size()` adjacent objects of the `value_type`

Should this concept be strengthened to the Standard C++ definition of the `Container`, there would necessarily be added an additional overload to each function accepting the `Container` with a generic `array_view` parameter type instead.

The following constructors and an operator are available for the `array_view`.

```
constexpr array_view() noexcept;
template <typename Container>
constexpr explicit array_view(Container& cont) noexcept;
template <typename ArrayType>
constexpr explicit array_view(ArrayType& data) noexcept;
template <typename ViewValueType>
constexpr array_view(const array_view<ViewValueType, rank>& rhs) noexcept;
```
template <typename Container>
constexpr array_view(bounds_type bounds, Container& cont) noexcept;
constexpr array_view(bounds_type bounds, pointer data) noexcept;

template <typename ViewValueType>
array_view& operator=(const array_view<ViewValueType, rank>& rhs) noexcept;

The default constructor creates an empty view – bounds() is zero for all components and data() is nullptr.

The single parameter Container constructor shall not participate in overload resolution unless the Container template argument satisfies the aforementioned Container concept. It is allowed only for rank = 1, what enables the view bounds to be deduced from the container size(). The view is created over the container data(). Note this constructor also allows for converting array_views with rank > 1 to array_views with rank = 1, i.e. flattening.

For example:
```cpp
auto vec = vector<int>(10);
auto av2 = array_view<int, 2>{{ 2, 5 }, vec }; // 2D view over vec
auto av1 = array_view<int, 1>{ vec }; // 1D view over vec
auto avf = array_view<int, 1>{ av2 }; // flattened av2; equivalent to av1
```

The single parameter ArrayType constructor shall not participate in overload resolution unless the following expression is true:
```cpp
is_convertible<add_pointer_t<remove_all_extents_t<ArrayType>>, pointer>::value && rank<ArrayType>::value == rank.
```
Scilicet, the ArrayType must be an array type with the value_type underlying type and the same rank as the view. The view will be created over the array data with the bounds derived from the array type extents.

For example:
```cpp
char r[3][1][2];
array_view<char, 3> av{ r }; // av.bounds() is {3, 1, 2}
```

The single parameter array_view constructor shall not participate in overload resolution unless is_convertible<add_pointer_t<ViewValueType>, pointer>::value is true. This overload serves as a copy constructor but also allows for implicit conversions between related array_view types, e.g. converting a view over mutable data to a view over constant data. The resulting view adopts the shape and the location of the original, effectively changing only the value_type.

The two-parameter constructor with the bounds_type and the Container parameters shall not participate in overload resolution unless the Container template argument satisfies the aforementioned Container concept. The view with the specified bounds is created over the provided container. There is a precondition that the container size() must be greater than or equal to the bounds size(). Since the array_view meets the Container requirements, this constructor may be used for reshaping the view with the same or different rank.

The two-parameter constructor with the bounds_type and the pointer to the value_type parameters has a precondition that the pointed to storage contains at least as many adjacent objects as the bounds size(). The view with the specified bounds is created over the pointed to collection.

The assignment operator is analogous to the single parameter array_view constructor.
strided_array_view Construction and Assignment

We have decided against providing a rich set of constructors for the strided_array_view similarly to the array_view rich set of constructors. It is based on the assumption that the path of least resistance should guide users to the more versatile array_view. The following constructors and an operator are available for the strided_array_view.

```cpp
constexpr strided_array_view();
template <typename ViewValueType>
constexpr strided_array_view(const array_view<ViewValueType, rank>& rhs) noexcept;
template <typename ViewValueType>
constexpr strided_array_view(const strided_array_view<ViewValueType, rank>& rhs) noexcept;
constexpr strided_array_view(bounds_type bounds, index_type stride, pointer data) noexcept;
template <typename ViewValueType>
strided_array_view& operator=(const strided_array_view<ViewValueType, rank>& rhs) noexcept;
```

The default constructor creates an empty view – all components of bounds() and stride() of the created object are zeroes.

The single parameter array_view constructor shall not participate in overload resolution unless is_convertible<add_pointer_t<ViewValueType>, pointer>::value is true. This constructor enables implicit conversion of an array_view to a strided_array_view, adopting the bounds, the stride and the location of the original object.

The single parameter strided_array_view constructor shall not participate in overload resolution unless is_convertible<add_pointer_t<ViewValueType>, pointer>::value is true. Alike the analogous array_view constructor, this function serves as a copy constructor and allows for conversion between related strided_array_view types.

The last constructor creates a view with all its properties explicitly specified. There are two preconditions – for any index idx, if bounds.contains(idx):

1) for i = [0, rank), idx[i] * stride[i] must be representable as ptrdiff_t; this is established in order to enable the implementation to use ptrdiff_t for the internal address calculation without overflowing
2) (*this)[idx] must refer to a valid memory location

We intentionally do not impose further restrictions on the stride, such as positivity or monotonicity in order to enable more advanced use cases such as self-aliasing or transposed views.

The assignment operator is analogous to the single parameter strided_array_view constructor.

Observers

The following observer functions are available both for the array_view and the strided_array_view.

```cpp
constexpr bounds_type bounds() const noexcept;
constexpr size_type size() const noexcept;
constexpr index_type stride() const noexcept;
```
`bounds()` returns the bounds of the view.

`size()` returns the size of the view, i.e. it is equivalent to `bounds().size()`.

`stride()` returns the stride of the view. The stride of an `array_view` is inferred from its bounds, following the logic drafted above for its collection requirements. Although this makes the function redundant for that type, it is provided nevertheless for the commonality of the two view types, which can be leveraged in a generic code.

**array_view Observers**
The following function is available only for the `array_view`.

```cpp
constexpr pointer data() const noexcept
```

The contiguity requirement of the `array_view` allows for the `data()` function, which returns a pointer to the storage over which the view was defined.

**Accessing Elements**
The following function is available both for the `array_view` and the `strided_array_view`. (See also: Appendix: Design Alternatives, X).

```cpp
constexpr reference operator[](const index_type& idx) const noexcept
```

The subscript operator allows to access the elements of the collection over which the view is created. The precondition for the function is that `bounds().contains(idx)`. The function returns a reference to an object which address is calculated using Equation 1, where: `data` is the result of `data()` for the `array_view` or the pointer supplied to the `strided_array_view` constructor (possibly implicitly) and stride is the result of `stride()`.

**Equation 1 array_view address calculation.**

\[
addr = data + \sum_{i=0}^{rank} idx[i] \times stride[i]
\]

Note that the subscript operator is a const function, even when returning a non-const reference. This is an intended design, akin to the pointer.

**Slicing**
Slicing is available for both the `array_view` and the `strided_array_view`.

```cpp
// For array_view<value_type, rank>:
array_view<value_type, rank - 1>
operator[](typename index_type::value_type slice) const noexcept;

// For strided_array_view<value_type, rank>:
strided_array_view<value_type, rank - 1>
operator[](typename index_type::value_type slice) const noexcept;
```

Slicing a view is an operation which returns a new view of an analogous type with a rank smaller by one. It is equivalent to “fixing” the most significant dimensions of the original view, as depicted on Figure 4. The function should not participate in overload resolution unless the rank of the view is greater than one. The precondition for the function is that the `slice` argument must be less than `bounds()[0]`. 
The syntax and semantics of the slicing operation allow also for cascading indexing, which enables patterns congruent with the “traditional” multidimensional addressing.

For example, for an appropriate 3-dimensional view `av` the two following lines are equivalent:

```cpp
av[1][2][3] = 42;
av[{1, 2, 3}] = 42;
```

**Sectioning**

Sectioning is available for both the `array_view` and the `strided_array_view`. (See also: Appendix: Design Alternatives, XI).

```cpp
strided_array_view<value_type, rank>
section(const index_type& origin, const bounds_type& section_bnd) const noexcept;
strided_array_view<value_type, rank>
section(const index_type& origin) const noexcept;
```

Sectioning a view is an operation which returns a new view of the same rank referring to the section of the original, as depicted on Figure 5. Since the contiguity of the result cannot be guaranteed, the returned object is always a `strided_array_view`. In the overload where the section bounds are not provided, they are assumed to extend to the remainder of the original view. The preconditions of the function require that for any index `idx`, if `section_bounds.contains(idx)`, `bounds().contains(origin + idx)` must be true (i.e. the newly created view is subsumed by the original).

**Comparison with the string_view**

The `string_view` proposal [3] introduces the titular `string_view` type, providing similar semantics to the `array_view` type from our proposal. In our view the proposals are complementary rather than competitive. Particularly, a `string_view` is to an `array_view` what a `string` is to an `array`. 
The single contention point we have identified is the decision in the *string_view* proposal to always assume the constness of the *basic_string_view*, by rationale that it is a more common case. Therefore the closest counterpart to a *basic_string_view* would be an *array_view<const char>* in our proposal. While we believe it may be confusing to users, at this point we cannot offer any alternative.

**bounds_iterator**

The *bounds_iterator* is provided as an interoperability feature, enabling the usage of the multidimensional indices with the existing non-mutable iterator-based algorithms. The *bounds_iterator* is dissimilar to other C++ Library iterators, as it does not perform iterations over containers or streams, but rather over an imaginary space imposed by the *bounds*. Dereferencing the iterator returns an *index* object designating the current element in the space.

Since the *bounds_iterator* provides the capability to traverse a multidimensional discrete space with the single dimensional iterator semantics, it is necessary to linearize the space. The iteration shall begin with an *index* which all coordinates are zeroes and increment the least-significant coordinate (i.e. *idx[rank - 1]*) first. Upon reaching the imaginary bound of the rectangular space in the given dimension (i.e. *bounds[rank - 1]*) the implementation shall wrap the least-significant coordinate around to 0 and increment the next more significant element. The process should be analogously applied to the more significant elements up to the point where the *index* reaches the largest value for all elements while contained within *bounds*, at which point subsequent increment reaches the past-the-end value (see visualization on Figure 6). The linearization of the *bounds_iterator* is congruent with the memory layout defined by *array_view*, thus using these two types together maintains the optimal access pattern.

![Figure 6 Visualization of bounds_iterator traversal over one- and two-dimensional bounds.](image)

The *bounds_iterator* is always a constant iterator, as there is no reasonable meaning to modifying the values of its current *index*.

Since the *bounds_iterator* is a proxy iterator, it cannot fulfill all requirements of the random access iterator concept. Specifically, the incompatibility is the inability to present a persistent object in the iteration space, which is surfaced as the *bounds_iterator*’s reference type (i.e. the return type of *operator*`) being a value type – *const index<Rank>*. Likewise, the result of *operator->* (of a pointer to *index<Rank>* type) must be considered invalidated after any operation on the iterator. We believe however that the discrepancy is small enough to be condoned, as it is expected not to cause any issues in the typical use cases. Furthermore, as a precedence, a similar divergence is often present in the implementation of the *std::vector<bool>::iterator*.

**Definitions**

The *bounds_iterator* is a template class with a single template parameter designating the rank of the *bounds* over which it iterates.
template <int Rank> class bounds_iterator;

Rank is required to be greater than zero, with the same rationale as for other types in the document.

The following iterator traits are provided as nested types.

using iterator_category = random_access_iterator_tag;
using value_type = const index<Rank>;
using difference_type = ptrdiff_t;
using pointer = const index<Rank>*;
using reference = const index<Rank>;

Construction and Assignment

explicit bounds_iterator(bounds<Rank> bnd, index<Rank> curr = index<Rank>{}) noexcept;
bounds_iterator(const bounds_iterator& rhs);

bounds_iterator& operator=(const bounds_iterator& rhs);

The first constructor creates an iterator over the specified bounds, with the initial position of the iterator set to the provided index. The precondition is that bnd.contains(curr) unless bnd.size() = 0.

The copy constructor and the copy assignment operator have trivial semantics of copying or assigning the bounds and the current index of the iterator.

Other Operations

The bounds_iterator may be obtained from the bounds object using the following namespace-scope functions or the corresponding member functions of the bounds type.

template <int Rank> bounds_iterator<Rank> begin(const bounds<Rank>& bnd) noexcept;

end returns an iterator over the provided bounds, with the initial position set to zero.

Possible Future Extensions

While we kept the proposed design to a minimal self-contained scope, our experience with C++ AMP suggests more versatile additions might be possible.

An array_view with an implicit data transfer and caching. Even though C++ does not recognize disjoint memory spaces within one computer system (which is often the case for the dedicated CPU and the dedicated discrete GPU memory), the growing trend in the end-user application programming is to leverage the compute resources of a remote computer system (viz. “the cloud”). While the problem is involved, one prerequisite is to accommodate for the required data transfer between the user system and the compute back-end, and vice-versa. We believe that the solution might be provided either by a novel type similar and related to the array_view or through an additional policy introduced on the array_view template. Note that C++ AMP has been providing the data transfer and the caching capabilities within one computer system since its first release in 2012.

A floating point-based addressing would be a continuous counterpart to the discrete addressing provided by the index. Such addressing is used primarily in computer graphics scenarios, where texture sampling (i.e. reading discrete container elements with filtering, giving the impression of continuity [4]) is pervasive.
It could be accommodated by a similar to index – float_index type, maybe along with a corresponding float_bounds type.

The bounds_iterator linearization in the current design is congruent with the memory layout imposed by the array_view, however there are other reasonable options, e.g. transposed. We believe that such alternatives could be provided as iterator adapters.

Prior Art
The Boost Multidimensional Array Library [5], apart from providing a multidimensional container template – which counterpart we purposefully did not aim to achieve in this proposal – offers multidimensional view capabilities as adapters for arrays of contiguous data. While the general concepts are similar to our proposal, the syntax and semantics details differ. Specifically: there are no novel types analogous to bounds and extent, but rather collections of scalar types or series of subscript operators are used; there is a separate type introduced to express views over constant data; semantics are not congruent with pointers, e.g. view’s assignment operator performs a deep copy; there is not a separate type for strided views.

array_ref, as described in “Proposing array_ref<T> and string_ref” [6], was the counterpart to the aforementioned string_view in an early version of the proposal. The type did not provide multidimensional capabilities.

valarray, defined in the Standard C++ Library, provides some view-like capabilities with slice_array and gslice_array classes, however they are defined only over valarray container. Furthermore there is no support for multidimensionality, apart from nesting the template types. Lastly, as these types are suited primarily for numeric calculations, we viewed extending them a less viable path than introducing our novel types.

Acknowledgments
David Callahan, Yossi Levanoni and Herb Sutter have designed the original C++ AMP interfaces.
Gabriel dos Reis, Artur Laksberg and Gor Nishanov have contributed to the final shape of this proposal.

References


Appendix: Design Alternatives

Names
We are fond of the names we have settled on in C++ AMP: extent and index. Unfortunately the name extent is already used in the Standard C++ Library (as a type trait for querying array type extents). One noteworthy alternative we have considered was indexes — denoting the fact that the object of such type is “a bag of index objects”, this option was abandoned after we realized how confusing the name is in conversation — “does ‘indexes’ mean two index objects or one indexes object”.

bounds and index
I. The bounds represent the axis-aligned N-dimensional rectangle with the minimum point at 0. An alternative representation would be one with an arbitrary origin (minimum point). We have decided against such design, because it would double the size requirement of the type, and more importantly — we have tried this approach in the Developer Preview version of C++ AMP (the type was then called grid) and received negative feedback as being too complex. If necessary, users can easily emulate a similar behavior by using bounds alongside with an index designating the desired offset.

II. We have explored a design alternative to the dual bounds and index types, where three distinct types were introduced instead — bounds<N> as an N-tuple of size_t for defining bounds, index<N> as an N-tuple of size_t for indexing elements and index_diff<N> as an N-tuple of ptrdiff_t for difference calculation or applying an offset. We found the resulting type proliferation daunting and it still posed the mentioned below (IV) problems of mismatching signed and unsigned component types in user programs.

Another alternative design might have called for a single type to represent both the bounds and the index. Although we agree that it would be a valid approach, we prefer the one where types of different responsibilities and semantics are clearly separated. Specifically, it would be impossible to express the N-dimensional rectangle and the N-dimensional vector semantics of the bounds and the index, which we view as important type safety features.

The corollary of the above reasoning is that any generic type (e.g. array, tuple) cannot substitute the bounds and the index either, unless they would be defined with a distinctive element type (e.g. array<index_t, 2> instead of index<2>), allowing for discriminating them for the relevant
operations. In spite of the fact that a similar approach to introducing new types is common in some functional languages, we view it as inconsistent with the common practice in C++.

III. The index and the bounds value_type is ptrdiff_t. An alternative approach would be to expose the type as a template type parameter of the index and the bounds. We have deemed it undesirable as this would preclude the secondary contribution of the proposal – providing a standard multidimensional indexing type.

IV. The bounds value_type is ptrdiff_t. An alternative approach would be to define the type as size_t. We have decided against it, as the index and the bounds are almost always used together and mismatching signed and unsigned component types would complicate user programs with conversions and/or warning noise.

V. Implicit construction of an index or bounds from value_type is allowed only for rank = 1. An alternative design would have allowed construction from a single value_type for any rank. There is however no agreement in the existing practice whether the omitted components should be filled with the same replicated value or with zeroes, thus it is the least confusing to provide neither.

VI. The initializer_list constructor available for both the index and the bounds requires the initializer list’s size to be equal to the rank. We view the fact that the precondition cannot be enforced by the type system as unfortunate. We have explored an alternative design where the constructor parameter type would be const array<value_type, rank>& instead of the initializer list, which would address half of the concern by enforcing the upper bound on the number of arguments (i.e. cannot be more than the rank). However the prevailing downside of this approach is a syntax quirk requiring the brace initialization of such type to be expressed with two pairs of braces (i.e. alt_index<2> idx {{1,2}}).

VII. An alternative design of the increment and the decrement operators for the bounds and the index was used in the C++ AMP library, where these operators were supported for any rank, performing the corresponding operation element-wise on all components – effectively traversing the space diagonally. We found this in practice to be more often surprising than useful.

array_view and strided_array_view

VIII. A design alternative to the dual array_view and strided_array_view types was used in the C++ AMP library, where there was a single type with the relaxed requirements (alike the strided_array_view). We have chosen to isolate the contiguous case in the current proposal on the grounds that: 1) on modern computer systems the memory access pattern is crucial for performance and as such should be explicit, 2) the guarantee on contiguity allows us to specify for array_view a safe data() member function, which enables interoperability with dimensionality-unaware algorithms (including C functions).

IX. The observation that the array_view is a special case of the strided_array_view may lead to an alternative design, where the array_view is derived from the strided_array_view. We have decided against such composition as the current design allows for more space-efficient
implementation of the `array_view` (its stride can be implied from its bounds). With the implicit conversion to the `strided_array_view` available, we believe the usability should not be severely handicapped.

X. A possible extension to the current `array_view` and the `strided_array_view` design offering `operator[]` would be to provide `at(const index_type&)` function, similarly to the `std::vector`. Such function could be throwing an exception when the supplied index were out-of-bounds. We are averse to this idea as it would conflict with the principle of fast-failing on programmer’s error, which we assumed throughout the design.

XI. For the `array_view` and the `strided_array_view`, an overload of `section` function with a single `bounds` parameter and the origin assumed to be 0 is not provided, as it would conflict with the single parameter `index` overload when the caller is using uniform initialization (e.g. `av.section({1, 2})`), and is easily achieved by using the first overload with an empty initializer for the origin (i.e. `av.section({}, {1, 2})`).