TUESDAY SEPTEMBER 13TH, 2022

CPPCON 2022

AURORA, COLORADO

MDSPAN
A DEEP DIVE SPANNING C++, KOKKOS & SYCL

NEVIN ":-)" LIBER
Computer Scientist
nliber@anl.gov

U.S. DEPARTMENT OF ENERGY
Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.

Argonne NATIONAL LABORATORY

WHO AM I?

WHO AM I?

- Argonne National Laboratory
  - Computer Scientist
    - Argonne Leadership Computing Facility
  - Kokkos, C++, SYCL
  - Aurora
  - C++ Committee
    - Vice Chair, Library Evolution Working Group Incubator (LEWGI / SG18)
  - Khronos SYCL Committee Member

WHAT IS MDSPAN?

MDSPAN

- `mdspan` is a *non-owning* multidimensional array view for C++23

- Vocabulary type
  - Usage in interfaces
  - Usage across domains

MDSPAN

- `mdspan` is a *non-owning* multidimensional array view for C++23

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class AccessorPolicy = default_accessor<ElementType>>
struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
}
```

MDSPAN

ElementType

- `mdspan` is a *non-owning* multidimensional array view for C++23

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class AccessorPolicy = default_accessor<ElementType>>
struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
}
```

MDSPAN

`ElementType`

- The elements in the array

MDSPAN

Extents

- `mdspan` is a *non-owning* multidimensional array view for C++23

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class AccessorPolicy = default_accessor<ElementType>>
struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
}
```

MDSPAN

`Extents`

- `Extents` describes the dimensions of the multidimensional array

MDSPAN

Extents

- `Extents` describes the dimensions of the multidimensional array

```
template<class IndexType, size_t... Es>
class extents;

template<class IndexType, size_t Rank>
using dextents = see below;
```

MDSPAN

Extents

- `Extents` describes the dimensions of the multidimensional array

```
template<class IndexType, size_t... Es>
class extents;
```

```
template<class IndexType, size_t Rank>
using dextents = see below;
```

MDSPAN

Extents

- `Extents` describes the dimensions of the multidimensional array

```
template<class IndexType, size_t... Es>
class extents;
```

MDSPAN

IndexType

- `Extents` describes the dimensions of the multidimensional array

```
template<class IndexType, size_t... Es>
class extents;
```

- The type used for the index (`int`, `size_t`, etc.)

MDSPAN

Es...

- `Extents` describes the dimensions of the multidimensional array

```
template<class IndexType, size_t... Es>
class extents;
```

- Each dimension
  - std::`dynamic_extent` if the dimension is determined at runtime
  - Any other number is the (compile-time) static dimension
- `Es...` are `size_t` because `std::dynamic_extent` is `size_t`

MDSPAN

dextents

- `Extents` describes the dimensions of the multidimensional array

```
template<class IndexType, size_t... Es>
class extents;
```

```
template<class IndexType, size_t Rank>
using dextents = see below;
```

MDSPAN

dextents

```
template<class IndexType, size_t Rank>
using dextents = see below;

dextents<int, 3>
```

- is an alias for

```
extents<int, dynamic_extent, dynamic_extent, dynamic_extent>
```

- Corentin Jabot & I really tried to get `std::dynamic_extent` shortened to `std::dyn` in C++20

MDSPAN

LayoutPolicy

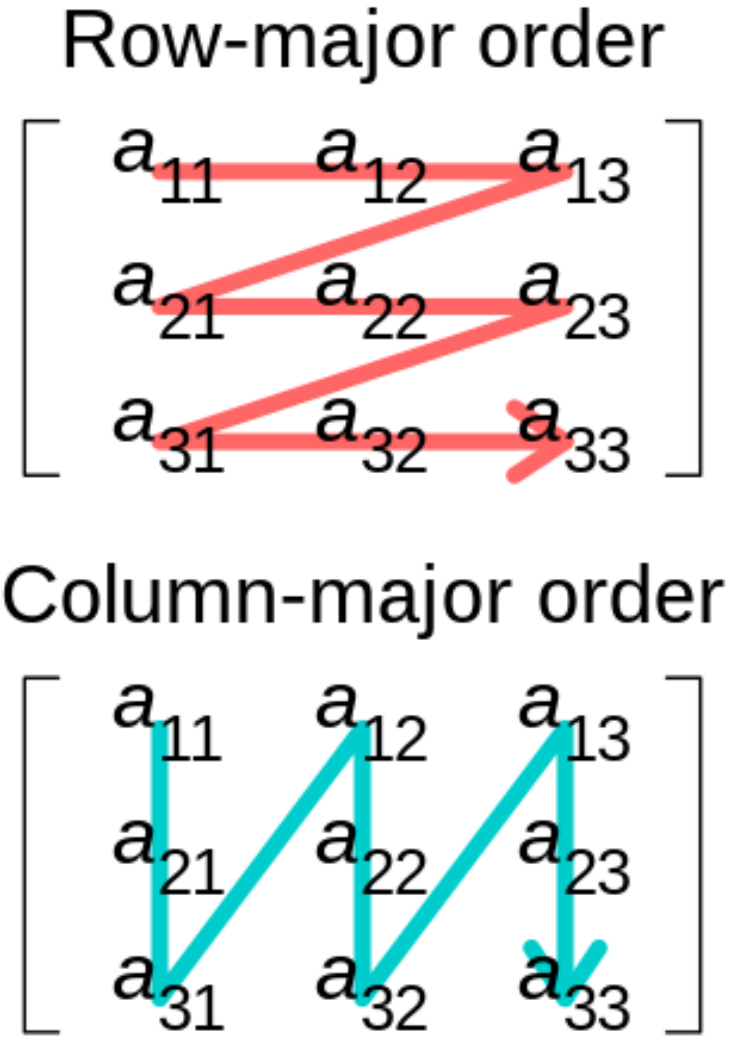
- mdspan is a *non-owning* multidimensional array view for C++23

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class AccessorPolicy = default_accessor<ElementType>>
struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
}
```

MDSPAN

LayoutPolicy

- Maps indices into offsets
  - `layout_right`
    - Rightmost extent is contiguous
      - `A[I, j] === A[i * M + j]`
    - Default
    - Row-major
    - C++ / C ordering
  - `layout_left`
    - Leftmost extent is contiguous
      - `A[i, j] === A[i + j * N]`
    - Column-major
    - Fortran ordering
  - `layout_stride`
  - User-defined
    - Tiled, Symmetric, Sparse, Compressed, etc.



MDSPAN

LayoutPolicy::mapping

```
template<typename Extents>
struct layout_*::mapping {
    //...
    constexpr const extents_type& extents() const noexcept;
    constexpr index_type required_span_size() const noexcept;

    template<class... Indices>
    constexpr index_type operator()(Indices...) const noexcept;

    static constexpr bool is_always_unique() noexcept;
    static constexpr bool is_always_exhaustive() noexcept;
    static constexpr bool is_always_strided() noexcept;

    static constexpr bool is_unique() noexcept;
    static constexpr bool is_exhaustive() noexcept;
    static constexpr bool is_strided() noexcept;

    constexpr index_type stride(rank_type) const noexcept;

    template<class OtherExtents>
    friend constexpr bool operator==(const mapping&, const mapping<OtherExtents>&) noexcept;
};
```

MDSPAN

AccessorPolicy

- `mdspan` is a *non-owning* multidimensional array view for C++23

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class AccessorPolicy = default_accessor<ElementType>>
struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
}
```

MDSPAN

`AccessorPolicy`

- Customize the `pointer` and `reference` types
- Add decorations like `restrict`
- Remote memory
- Compressed memory
- Atomic access
  - `std::atomic_ref`

MDSPAN

default_accessor

```
template<class ElementType>
  struct default_accessor {
    using offset_policy =    default_accessor;
    using element_type =     ElementType;
    using reference =        ElementType&;
    using data_handle_type = ElementType*;

    constexpr default_accessor() noexcept = default;

    template<class OtherElementType>
    constexpr default_accessor(default_accessor<OtherElementType>) noexcept {}

    constexpr reference access(data_handle_type p, size_t i) const noexcept { return p[i]; }

    constexpr pointer   offset(data_handle_type p, size_t i) const noexcept { return p + i; }
  };
```

MDSPAN

AccessorPolicy

- `mdspan` is a *non-owning* multidimensional array view for C++23

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class AccessorPolicy = default_accessor<ElementType>>
struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
}
```

- Construct it with a pointer and a list of extents

MDSPAN

AccessorPolicy

- `mdspan` is a *non-owning* multidimensional array view for C++23

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class AccessorPolicy = default_accessor<ElementType>>
struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
}
```

- Index it via *m*[*2*, *3*, *5*]

MDSPAN

- `mdspan` is a *non-owning* multidimensional array view for C++23

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class AccessorPolicy = default_accessor<ElementType>>
struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
}
```

HOW DID WE GET HERE?

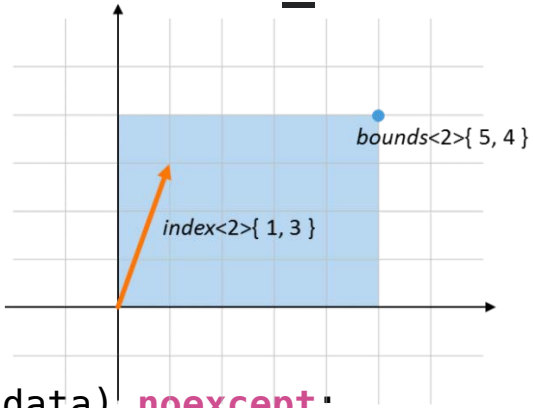
HOW DID WE GET HERE?

AN EIGHT YEAR MISSION…

N3851 MULTIDIMENSIONAL BOUNDS, INDEX AND ARRAY_VIEW

Łukasz Mendakiewicz & Herb Sutter (Microsoft)

- Based on C++Amp
- Only static extents




```
template <class ValueType, int Rank = 1>
struct array_view {
    constexpr array_view(bounds<Rank> bounds, ValueType* data) noexcept;
    // ...
    constexpr reference operator[](const index<Rank>& idx) const noexcept;
}
```

- `strided_array_view`
  - Contiguity in the least significant dimension is lifted

ARRAY_VIEW

Issaquah 2014

- Would like variadic `operator[]` but don't want to wait for language support
- Would like to mix static and dynamic extents

ARRAY_VIEW

Issaquah 2014

- Polls: *Strongly Favor* | *Weakly in Favor* | *Neutral* | *Weakly Against* | *Strongly Against*
  - Comfortable with `[(1, 2)]` syntax? 6 4 2 2 1
  - Comfortable with `(1, 2)` syntax instead of `[()]`? 3-2-3-5-1
  - Comfortable with 2 spellings? 0 1 3 1 10
  - Delay paper in ArraysTS until `fixed_array_view`? 0 0 4 4 8
  - `array_view` should have iterators with `ValueType array_view`? 0 0 3 7 4
  - Add a layout template parameter? 5 6 5 1 0
  - Hold up ArraysTS for layout? Unanimously NO
  - **Take `array_view` for ArraysTS? 9 5 1 0 0**

THANKS FOR LISTENING!

THANKS FOR LISTENING!*

THANKS FOR LISTENING!*

*IF ONLY…

SIDEBAR: ARRAYS TS

ARRAYS TS

Born 2013 (Chicago)

- Runtime-sized arrays with automatic storage duration
  - Stack arrays
  - No bounds checking, overrun the stack, etc.
  - Need a safe way to access it, iterators, etc.

CONTIGUOUS TYPES & CONTAINERS

Type	***Models***	***Min Capacity***	***Max Capacity***
`T`	Exactly 1	Compile Time	Compile Time
`optional<T>`	Up to 1	Compile Time	Compile Time
`array<T,N>`	Exactly N	Compile Time	Compile Time
`dynarray<T>`	Exactly N	Run Time	Run Time
~~*`static_vector<T,N>`*~~ ~~*`fixed_capacity_vector<T,N>`*~~ *`static_vector<T,N>`*	Up to N	Compile Time	Compile Time
~~*`clump<T,N,A>`*~~ *`small_vector<T,N,A>`*	Indefinite	Compile Time	Run Time
`vector<T,A>`	Indefinite	Run Time	Run Time

CONTIGUOUS TYPES & CONTAINERS

Type	*Models*	*Min Capacity*	*Max Capacity*
`T`	Exactly 1	Compile Time	Compile Time
`optional<T>`	Up to 1	Compile Time	Compile Time
`array<T,N>`	Exactly N	Compile Time	Compile Time
`dynarray<T>`	*Exactly N*	*Run Time*	*Run Time*
~~*`static_vector<T,N>`*~~ ~~*`fixed_capacity_vector<T,N>`*~~ *`static_vector<T,N>`*	Up to N	Compile Time	Compile Time
~~*`clump<T,N,A>`*~~ *`small_vector<T,N,A>`*	Indefinite	Compile Time	Run Time
`vector<T,A>`	Indefinite	Run Time	Run Time

ARRAYS TS

- Need a safe way to access it, iterators, etc.
  - `class dynarray`
    - `Allocator` not part of the type
      - Passed to constructors
    - How does *dynarray* "stack memory" work if embedded in another type?
      - And what if that aggregate type is on the heap?
      - Compiler writers do not know how to implement this

- ***Impasse!***
  - We'll have an Arrays TS (Technical Specification) to sort it out

ARRAYS TS

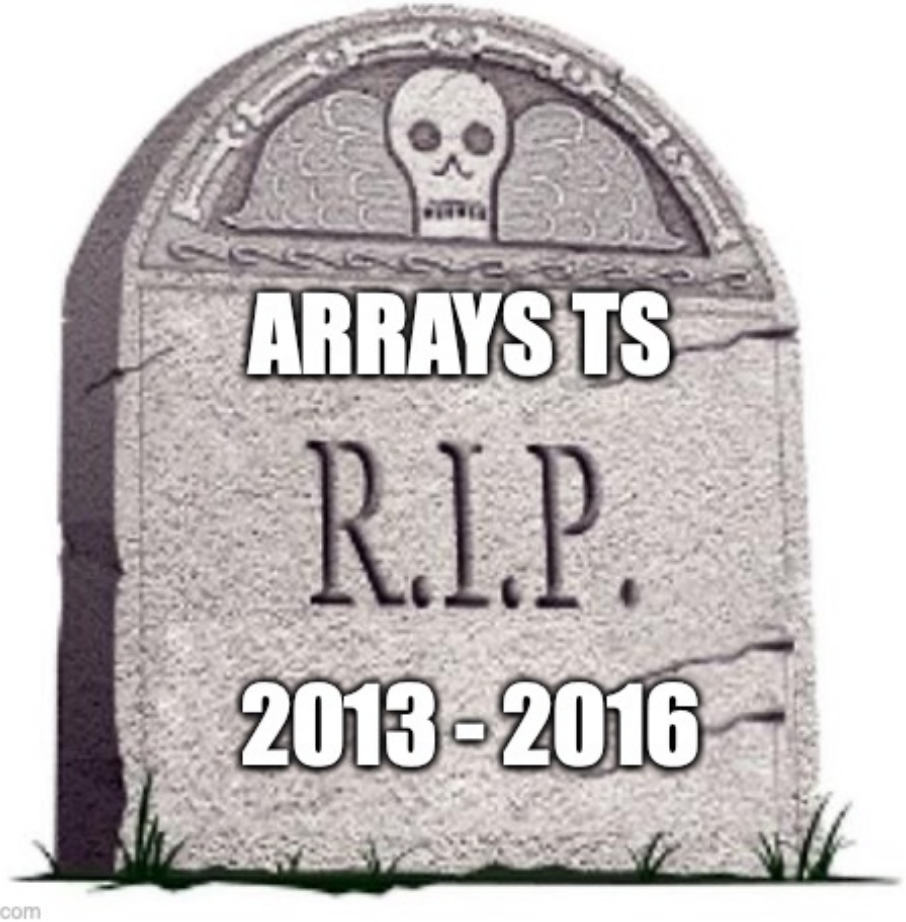
2014 (Issaquah)

- Let us add all the array like things
  - `array_view`
  - `array_ref` (span-like) and `string_ref` (`string_view`)
  - Multi-dimensional support to `std::array`
  - `make_array`
  - Extend `shared_ptr` and `make_shared` to support arrays

ARRAYS TS

Died 2016 (Jacksonville)

- Should we kill the Arrays TS?
  - 8 5 6 0 0



BACK TO 2014 AND ARRAY_VIEW

ARRAY_VIEW

N3976 (Revision 2) - Rapperswil

- Wording
- Some minor changes
- Send to Library Fundamentals TS v2
  - **10 4 1 2 0**

ARRAY_VIEW

N4087 (Revision 3)

- Minor fixes

OTHER IMPORTANT C++ STUFF AROUND THIS TIME

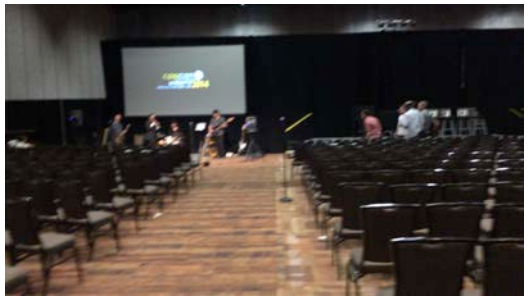
THE FIRST CPPCON

September 7th-12th, 2014 Bellevue, Washington

Happy to be there…

So young…

Making history…



Energetic…

THE FIRST CPPCON

September 7th-12th, 2014 Bellevue, Washington

THE FIRST CPPCON

September 7th-12th, 2014 Bellevue, Washington

But I digress…

ARRAY_VIEW

N4177 (Revision 4) - Urbana

- Forward to formal motions?
  - **5 4 2 0 0**

N4222 MINIMAL ADDITIONS TO THE ARRAY VIEW LIBRARY FOR PERFORMANCE AND INTEROPERABILITY

Rutger ter Borg & Jesse Perla

- Urbana (after meeting)
  - `StorageOrder` template tag (layout)
  - `fixed_array_view` (static extents as template parameters)
  - `StrideType` template parameter (layout)
  - Variadic `operator()` for index lookup
  - Polls
    - Split 1D `array_view` from multidimensional one 1 1 4 5 0
    - If variadic `operator[]` comes along, use it (else `operator()`) 6 3 2 0 0
    - Allow `operator[]` for 1D case 4 4 3 0 0

D4300 ISSUES WITH ARRAY_VIEW

H. Carter Edwards (Sandia / Kokkos)

- Array layout has a significant impact on performance, including simd-vectorize
- Array layout performance considerations should include tiling and padding
- Strided should be a layout option
- Ability to mix compile-time and runtime dimensions has performance impact
- index and bounds could be replaced by `std::array<ptrdiff_t, Rank>`
- `array_view<T, Rank>` inconsistent with `std::array<T, Length>`
- Interaction with memory management is not addressed

- Recommendation: delay N4177 to ArraysTS

2015

ARRAY_VIEW

N4346 (Revision 5) - Cologne (L(E)WG only meeting)

- LWG changes requested in Urbana incorporated
- Many detailed comments
- **“Looking in good shape to move in Lenexa”**

ARRAY_VIEW

N4494 (Revision 6) - Lenexa

- LWG changes requested in Cologne incorporated

ARRAY_VIEW

N4512 (Revision 7) - Lenexa

- LWG changes requested in Lenexa (earlier in the week) incorporated
- Formal motions straw polls page
  - Voted down

KOKKOS::VIEW

KOKKOS::VIEW

- Multi-dimensional array of zero or more dimensions

```
template <class DataType
          [, class LayoutType]
          [, class MemorySpace]
          [, class MemoryTraits]>
class View;
  .
```

KOKKOS::VIEW

DataType

- Multi-dimensional array of zero or more dimensions

```
template <class DataType
          [, class LayoutType]
          [, class MemorySpace]
          [, class MemoryTraits]>
class View;
  .
```

KOKKOS::VIEW

DataType

- Multi-dimensional array of zero or more dimensions

```
template <class DataType
          [, class LayoutType]
          [, class MemorySpace]
          [, class MemoryTraits]>
class View;
  .
```

KOKKOS::VIEW

DataType

- Runtime (dynamic) and compile time (static) dimensions
- Const views
- Terse notation that must be valid C++ syntax
  - Requires runtime dimensions appear first
  - `View<double**>`
    - 2D `View` of `double` with 2 runtime dimensions
  - `View<const int***[5][3]>`
    - 5D `View` of `int` with 3 runtime and 2 compile time dimensions.
    - The data is read-only (`const`).

KOKKOS::VIEW

- Index with `operator()`
  - 1D Views can be indexed with `operator[]`
- *Sometimes* owning
  - (Kokkos parlance: managed or unmanaged)
- *Sometimes* reference counting
  - Not inside `parallel_for`, `parallel_reduce`, `parallel_scan`

KOKKOS::VIEW

LayoutType

- Multi-dimensional array of zero or more dimensions

```
template <class DataType
          [, class LayoutType]
          [, class MemorySpace]
          [, class MemoryTraits]>
class View;
```

  - Maps indices into offsets
    - `LayoutLeft`, `LayoutRight`, `LayoutStride`, `LayoutTiled`

KOKKOS::VIEW

MemorySpace

- Multi-dimensional array of zero or more dimensions

```
template <class DataType
          [, class LayoutType]
          [, class MemorySpace]
          [, class MemoryTraits]>
class View;
```

  - Where the memory resides
    - CPU, GPU, etc.

KOKKOS::VIEW

MemoryTraits

- Multi-dimensional array of zero or more dimensions

```
template <class DataType
          [, class LayoutType]
          [, class MemorySpace]
          [, class MemoryTraits]>
class View;
```

KOKKOS::VIEW

MemoryTraits

- AccessorPolicy + Managed/Unmanaged
- Atomic
- RandomAccess
  - Hint
- C `restrict`

KOKKOS::VIEW

Under the covers

- Multi-dimensional array of zero or more dimensions

```
template <class DataType
          [, class LayoutType]
          [, class MemorySpace]
          [, class MemoryTraits]>
class View;
```

KOKKOS::VIEW

Under the covers

- Multi-dimensional array of zero or more dimensions

```
template <class DataType
          [, class LayoutType]
          [, class MemorySpace]
          [, class MemoryTraits]>
class View;

template <class DataType, class... Properties>
class View;
```

BACK TO 2015 - LENEXA

N4355 : SHARED MULTIDIMENSIONAL ARRAY WITH POLYMORPHIC LAYOUT

H. Carter Edwards & Christian Trott

```
template<class ArrayType,
         class ArrayLayout = void,
         class SizeType = size_t>
class shared_array;

template<class ArrayType,
         class ArrayLayout = void,
         class SizeType = size_t>
class weak_array;
```

N4355 : SHARED MULTIDIMENSIONAL ARRAY WITH POLYMORPHIC LAYOUT

`SizeType`

- Being able to customize the `SizeType` is important for performance!

N4355 : SHARED MULTIDIMENSIONAL ARRAY WITH POLYMORPHIC LAYOUT

Lenexa

- `ArrayType`

$$T[\ N0_{opt}\ ][\ N1_{opt}\ ][\ N2_{opt}\ ]$$

  - Language change!

- Allow `operator[]` for rank-1?
  - 2 6 1 2 1

P0122R0 ARRAY_VIEW: BOUNDS-SAFE VIEWS FOR SEQUENCES OF OBJECTS

Neil MacIntosh (Microsoft)

- Pub Quiz!
  - This is the first appearance of

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

  - Bonus question: what are the differences between this and `byte` in C++17?

```
enum class byte : unsigned char {};
```

    - Type punning is allowed

- Constructing with invalid values
  - `std::terminate`
    - Not undefined behavior

P0122R0 ARRAY_VIEW: BOUNDS-SAFE VIEWS FOR SEQUENCES OF OBJECTS

Neil MacIntosh (Microsoft)

```
// class that represents a point in a multidimensional space
template <size_t Rank, typename ValueType = size_t>
class index;

// 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;

template <size_t Rank, typename SizeType = size_t>
class strided_bounds;
```

P0122R0 ARRAY_VIEW: BOUNDS-SAFE VIEWS FOR SEQUENCES OF OBJECTS

Neil MacIntosh (Microsoft)

```
// 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;

template <>
struct dim<dynamic_range>;
```

P0122R0 ARRAY_VIEW: BOUNDS-SAFE VIEWS FOR SEQUENCES OF OBJECTS

Neil MacIntosh (Microsoft)

```
// 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;
	};
};
```

P0122R0 ARRAY_VIEW: BOUNDS-SAFE VIEWS FOR SEQUENCES OF OBJECTS

Neil MacIntosh (Microsoft)

```
// 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;
```

P0122R0 ARRAY_VIEW: BOUNDS-SAFE VIEWS FOR SEQUENCES OF OBJECTS

My thoughts

- First time I can remember being present for an `array_view` discussion
- Way too complicated for mere mortals like me
- I'll let the people who need this figure it out

- *If only I knew I would be entering this field exactly three years later to the day!…*

P0122R1 SPAN: BOUNDS-SAFE VIEWS FOR SEQUENCES OF OBJECTS

- Removed multidimensional aspects from the proposal
- Unfortunately, I stopped paying attention to this proposal for a while…

SIDEBAR: SPAN

SPAN

```
using size_type = size_t;
```

- I’m sorry

SPAN

```
using size_type = size_t;
```

- I’m sorry
  - I lead the charge
    - Up against well-known C++ luminaries
    - At least two of which were on that Committee Fireside Chat stage last night
  - Before I was in HPC
    - And thought the performance differences were minor
      - *Which is what committee members say when they want a feature*
      - Don’t want a feature? *Can’t afford even one cycle*
  - More important was Interoperability with the rest of the standard library

P0009

P0009R0 POLYMORPHIC MULTIDIMENSIONAL ARRAY VIEW

H. Carter Edwards, Christian Trott, Juan Alday (finance), Jesse Perla (UBC.CA), Mauro Bianco (CSCS.CH), Robin Maffeo (AMD), Ben Sander (AMD), Bryce Lelbach (LBL)

- Not just Kokkos folks
- The essential issue with `array_view`
  - Did not fulfill C++'s zero-overhead abstraction
    - For both static and dynamic extents
    - Different memory layouts
      - Eigen
      - Matlab's C++ interface
  - Would need another library for "direct mapping to the hardware"

P0009R0 POLYMORPHIC MULTIDIMENSIONAL ARRAY VIEW

- Layout more general
  - Different orderings
  - Padding
- Interoperability with libraries using compile-time extents
- Zero-overhead abstraction for `constexpr` extents and strides
- Extensibility for view properties beyond dimensions and layouts

P0009R0 POLYMORPHIC MULTIDIMENSIONAL ARRAY VIEW

Multiple implicit dimensions

```
template<class DataType, class... Properties>
struct view;
```

- View of multidimensional array with multiple implicit dimensions
  - Either pass a property, or…
  - “Requires slight language specification change for correction and relaxation of array declaration.”

```
view<int, view_property::implicit_dimensions<3>>;
view<int[][][]>;
```

P0009R0 POLYMORPHIC MULTIDIMENSIONAL ARRAY VIEW

Multiple implicit dimensions

```
view<int, view_property::implicit_dimensions<3>>;
view<int[][][]>;
```

- *Equivalent-but-distinct* types
- Issues when declaring the type in an interface

```
void DoSomething(view<???> v);
```

- Separate overloads
- Pay (small) runtime conversion cost
- Stay in template-land

P0009R0 POLYMORPHIC MULTIDIMENSIONAL ARRAY VIEW

- Layout
  - `view<int[][][], view_property::layout_left>`

- (Variadic) Properties get you flexibility and extensibility
  - At the cost of many *equivalent-but-distinct* types

- Polls
  - Do we want static zero-length extents? 3 4 2 3 0
  - Do we want property lists in the template arguments? 3 6 3 0 0
  - Do we want per view bounds checking? 3 4 2 1 1

P0009R0 POLYMORPHIC MULTIDIMENSIONAL ARRAY VIEW

Kona 2015

- Bikeshed
  - `array_ref`
- What about errors?
  - Contracts?

SIDEBAR: CONTRACTS

CONTRACTS

- C++ has only one knob that says “Here Be Dragons”
  - ***Undefined Behavior***
  - Everything else is *defined* behavior
    - And developers *will* write code dependent on defined behavior
- Contracts will give us more knobs
  - C++26 hopefully?

BACK TO P0009

P0009R1 POLYMORPHIC MULTIDIMENSIONAL ARRAY REFERENCE

Jacksonville 2016

- view is now `array_ref`
- Debate on signed vs. unsigned `size_type`

P0009R2 POLYMORPHIC MULTIDIMENSIONAL ARRAY REFERENCE

pre-Oulu 2016

- Add details for layout mapping
- Relaxed array declaration syntax moved to P0332
- Motivation and examples moved to P0331

P0009R3 POLYMORPHIC MULTIDIMENSIONAL ARRAY REFERENCE

post-Oulu 2016

- Undesirable Extent Mechanism (B) Proposal

```
template<size_t... IntegralExtent>
struct extents;
```

P0009R4 POLYMORPHIC MULTIDIMENSIONAL ARRAY REFERENCE

Albuquerque 2017

- Renamed to `mdspan`
- Align with `span`
- `extents` now part of this proposal
  - Still hoping for `mdspan<int[][][]>`

- Polls
  - We should be able to index with `span<`*`int-type`*`[N]>` (in addition to an array)? 2 11 1 1 0
  - We should be able to index with 1d `mdspan`? 0 8 7 0 0
  - **Forward this to LWG for Library Fundamentals v3? Unanimous consent**

LIBRARY FUNDAMENTALS V3

First working draft post-Rapperswil 2018

- Never got `mdspan`
- As for it shipping…
  - P2631R0 Publish TS Library Fundamentals 3 Now! - Alisdair Meredith, Bryce Adelstein Lelbach, Jonathan Wakely
    - September 15th, 2022 mailing *(two days from now!)* for Kona discussion
    - Other possible directions *(rejected by the authors)*:
      - Rebase onto C++23
      - Merge into C++23
      - Merge into C++26
      - Drop the TS entirely

P0009R5 POLYMORPHIC MULTIDIMENSIONAL ARRAY REFERENCE

Jacksonville 2018

- P0009R4 changes except `span<`*`int-type`*`[N]>` (weak support 2 11 1 0 0 & no proven need)
- P0009R5 not reviewed in Jacksonville

P0900R0 AN ONTOLOGY FOR PROPERTIES OF MDSPAN (DAISY HOLLMAN)

Jacksonville 2018

- We want the customization of `basic_mdspan` to be two customization points `Mapper` and `Accessor` (akin to `Allocator` design)?

```
basic_mdspan<T, Extents, Mapper, Accessor>
mdspan<T, N...>
```

  - 3 4 5 1 0
  - WA - I don't want too many types in the template argument list

- We want the customization of `basic_mdspan` to be an arbitrary (and potentially user-extensible) list of properties (akin to `Executor` property design)?

```
basic_mdspan<T, Extents, Properties...>
```

  - 1 2 2 6 2

P0009R6 MDSPAN: A NON-OWNING MULTIDIMENSIONAL ARRAY REFERENCE

Rapperswil 2018

- Replaced variadic property list with extents, layout mapping and accessor properties
- Added accessor policy concept
- Renamed `mdspan` to `basic_mdspan`

```
// Multidimensional span:
template <typename ElementType,
          typename Extents,
          typename LayoutPolicy = layout_right,
          typename AccessorPolicy = accessor_basic>
class basic_mdspan;

template <class T, ptrdiff_t... Extents>
using mdspan = basic_mdspan<T, extents<Extents...>>;
```

P0009R7 MDSPAN: A NON-OWNING MULTIDIMENSIONAL ARRAY REFERENCE

post-Rapperswil 2018

- Wording
- How to refer to `span` (as that will be in C++20, not C++17)

P0009R8 MDSPAN: A NON-OWNING MULTIDIMENSIONAL ARRAY REFERENCE

San Diego 2018

- Update based on reference implementation

P0009R9 MDSPAN: A NON-OWNING MULTIDIMENSIONAL ARRAY REFERENCE

Kona 2019

- Wording

- *(I joined Argonne a week before this meeting)*

P1161R3 DEPRECATE USES OF THE COMMA OPERATOR IN SUBSCRIPTING EXPRESSIONS - CORENTIN JABOT

Kona 2019 for C++20

CURRENT

```
array[x]      // Ok
array[(x,y)] // Ok, uses y as index/key
array[x,y]    // Ok, uses y as index/key
```

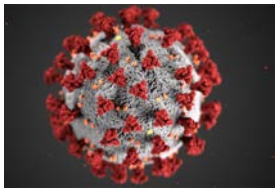
PROPOSED

```
array[x]      // Ok
array[(x,y)] // Ok, uses y as index/key
array[x,y]    // Deprecated,
              // uses y as index/key
```

P0009R10 MDSPAN: A NON-OWNING MULTIDIMENSIONAL ARRAY REFERENCE

Prague 2020

- Wording and paper cleanup
- C++20 is done!
  - `span` is hot
  - `mdspan` is not
- I accepted the LEWGI vice chair position
- And then the pandemic hit…



P0009R11 MDSPAN

2021 telecons, telecons, telecons…

- I'm now an author on P0009!

- Changed all sizes from `ptrdiff_t` to `size_t`
- Explicit about trivially copyable

TRIVIALLY COPYABLE

- How do we copy objects in C++?
  - Copy constructor / copy assignment operator
    - Running code
    - Code may access both source and destination
  - Can we do the same for inter-device copying (host/device or device/device)?
    - Where would the code run?
    - May not be able to simultaneously access source and destination
  - We can copy the bytes (object representation) that make up the object
  - C++ trivially copyable used as a proxy for types where we can copy the bytes

P0009R12 MDSPAN

2021 telecons, telecons, telecons…

- Now L(E)WG wants the design decisions *back* in the paper **sigh**
- Poll
  - Prefer the IS over LFTSv3 as ship vehicle for P0009 (`mdspan`)
  - 10 6 1 0 0

- Still hopeful for `mdspan<T[][][]>` `mdspan<T[][64][]>` syntax

P2128 MULTIDIMENSIONAL SUBSCRIPT OPERATOR

Mark Hoemmen, Daisy Hollman, Corentin Jabot, Isabella Muerte, Christian Trott

- Because P1161 deprecated the use of comma expressions in subscript expressions in C++20
  - Now make them ill-formed and give a new meaning to commas in subscript expressions
- `a[a, y, z]`
  - Eliminate workarounds
    - `a(x, y, z)`
    - `a[x][y][z]`
    - `a[{x, y, z}]`
- "We propose that `operator[]` should be able to accept zero or more arguments, including variadic arguments."
- "Both its use and definition would match that of `operator()`."
  - Except P1169 `static operator()` changes fell through the cracks
  - P2589 `static operator[]` would fix this, but probably not in time for C++23

P0009R13 MDSPAN

2021 telecons, telecons, telecons…

- `dextents` type alias
- Removed old `mdspan` and renamed `basic_mdspan` to `mdspan`

```
template<class ElementType, class Extents, class LayoutPolicy, class AccessorPolicy>
class basic_mdspan { /* ... */ };

template<class T, size_t... Extents>
  using mdspan = basic_mdspan<T, extents<Extents…>>;
```

- Deduction guides (CTAD) [P2299]
- `operator[]`

CLASS TEMPLATE ARGUMENT DEDUCTION (CTAD) (AKA DEDUCTION GUIDES)

CLASS TEMPLATE ARGUMENT DEDUCTION (CTAD)

Deduction Guides (C++17)

- Class template parameters are deduced from the constructor arguments

```
template<class T1, class T2>
pair(T1, T2) -> pair<T1, T2>;
```

- All parameters must be deduced
- Implicit and user defined ones
- Creates a *different* overload set
  - Exactly one match `pair(…)` -> exactly one match `pair<T1, T2>::pair(…)`

```
pair<int, const char*> t(2, "Three"); // All pair ctors
pair d(2, "Three");                   // Only CTAD ctors
```

CLASS TEMPLATE ARGUMENT DEDUCTION (CTAD)

- No need to specify template parameters when declaring non-member variables
  - *Immense gain in usability* for types with lots of template parameters
    - Like `mdspan`
  - But it is a tradeoff
    - Still need to know the exact type with template parameters
      - Declaring member variables
      - Compile time debugging

CLASS TEMPLATE ARGUMENT DEDUCTION (CTAD)

mdspan (from C++23)

```
template<class CArray>
  requires(is_array_v<CArray> && rank_v<CArray> == 1)
  mdspan(CArray&)
    -> mdspan<remove_all_extents_t<CArray>, extents<size_t, extent_v<CArray, 0>>>;

template<class Pointer>
  requires(is_pointer_v<remove_reference_t<Pointer>>)
  mdspan(Pointer&&)
    -> mdspan<remove_pointer_t<remove_reference_t<Pointer>>, extents<size_t>>;

template<class ElementType, class... Integrals>
  requires((is_convertible_v<Integrals, size_t> && ...) && sizeof...(Integrals) > 0)
  explicit mdspan(ElementType*, Integrals...)
    -> mdspan<ElementType, dextents<size_t, sizeof...(Integrals)>>;

template<class ElementType, class OtherIndexType, size_t N>
  mdspan(ElementType*, span<OtherIndexType, N>)
    -> mdspan<ElementType, dextents<size_t, N>>;

template<class ElementType, class OtherIndexType, size_t N>
  mdspan(ElementType*, const array<OtherIndexType, N>&)
    -> mdspan<ElementType, dextents<size_t, N>>;

template<class ElementType, class IndexType, size_t... ExtentsPack>
  mdspan(ElementType*, const extents<IndexType, ExtentsPack...>&)
    -> mdspan<ElementType, extents<IndexType, ExtentsPack...>>;

template<class ElementType, class MappingType>
  mdspan(ElementType*, const MappingType&)
    -> mdspan<ElementType, typename MappingType::extents_type,
              typename MappingType::layout_type>;

template<class MappingType, class AccessorType>
  mdspan(const typename AccessorType::data_handle_type&, const MappingType&,
         const AccessorType&)
    -> mdspan<typename AccessorType::element_type, typename MappingType::extents_type,
              typename MappingType::layout_type, AccessorType>;
```

CLASS TEMPLATE ARGUMENT DEDUCTION (CTAD)

mdspan(T*) // pointer to one object

```
template<class Pointer>
  requires(is_pointer_v<remove_reference_t<Pointer>>)
  mdspan(Pointer&&)
    -> mdspan<remove_pointer_t<remove_reference_t<Pointer>>, extents<size_t>>;

// …

int i = 0;
int* p = &i;
mdspan d(p);

// mdspan(int*&)
// remove_reference_t<int*&> --> int*
// is_pointer_v<int*> == true
// remove_pointer_t<int*> --> int
// mdspan<int, extents<size_t>> ->
   mdspan<int, extents<size_t>, layout_right, default_accessor<int>>
// template<class... OtherIndexTypes>
   constexpr explicit mdspan<int, extents<size_t>>::mdspan(int*, OtherIndexTypes... exts);
// template<> constexpr explicit mdspan<int, extents<size_t>>::mdspan(int*)
```

P0009

P0009R14 MDSPAN

2021 telecons, telecons, telecons…

- Send P0009R14 (`mdspan`) to LWG for C++23 with priority P3 (to be confirmed with a Library Evolution electronic poll)
  - **9 7 0 0 0**
  - 11/2021

P0009R15 MDSPAN

2022 telecons, telecons, telecons…

- LWG wording review

P0009R16 MDSPAN

2022 telecons, telecons, telecons…

- LWG wording review

P0009R17 MDSPAN

2022 telecons, telecons, telecons…

- `submdspan` moved to a separate paper
  - Not enough time to review it before C++23 feature freeze
  - I didn't have enough time to review it in this talk either. :-)

P2553R2 MAKE MDSPAN SIZE_TYPE CONTROLLABLE

2022 telecons, telecons, telecons…

- Added `SizeType` template parameter for `extents`

```
template<class SizeType, size_t... Es>
class extents;
```

  - Initially constrained to unsigned types
  - LEWG relaxed that constraint
    - Do not constrain extents size_type to unsigned_integral, allow for signed extents 7-8-1-0-0
    - The concept of the size_type should be a Mandate rather than a Constraint 7-10-0-0-0
    - Send [P2553R1] Make mdspan size_type Controllable to Library Working Group for C++23, classified as an improvement of an existing feature ([P0592R4] bucket 2 item) 7-9-1-1-0

P2599R2 INDEX_TYPE & SIZE_TYPE IN MDSPAN

2022 telecons, telecons, telecons…

- Throughout the standard, `size_type` stands for an *unsigned* type
- Rename `size_type` to `index_type`
- What should `mdspan::size()` return?
  - P0009R16 returned *old* `size_type`
  - P0009R17 returned `size_t`

```
template<...> class extents { // ...
    using size_type = make_unsigned_t<index_type>;
};

template<...> class mdspan { // ...
    using size_type = typename extents::size_type;
    constexpr size_type size() const noexcept;
};
```

- Separate paper from P0009 to lessen risk of P0009 not making C++23

P2599R2 INDEX_TYPE & SIZE_TYPE IN MDSPAN

2022 telecons, telecons, telecons…

- Send P2599R0 (mdspan::size_type should be index_type) to Library for C++23 classified as an improvement (B2), to be confirmed with a Library Evolution electronic poll 2-7-2-2-0
  - SA: It's already a conscious choice by the user to use a signed type. So I don't think it will be surprising. The consistency of having it be called size_type is more important
- mdspan, extents, and layouts should have both an index_type (which is whatever the user provides for the first template parameter to extents) and a size_type (which is make_unsigned_t<index_type>) 3-9-1-1-0
  - WA: It's additional complexity
- Modify P2599R1 (mdspan::size_type should be index_type) such that mdspan::sizes return type is size_type, and send the modified paper to Library for C++23 classified as B2 - Improvement, to be confirmed with a Library Evolution electronic poll 5-8-0-1-0
  - WA: This is a late change.
- Put P2599R2 into C++23 pending LEWG approval 18-0-0
- Send [P2599R2] index_type & size_type In mdspan to Library Working Group for C++23, classified as an improvement of an existing feature ([P0592R4] bucket 2 item) 14-7-2-1-0

P2604R0 MDSPAN: RENAME POINTER AND CONTIGUOUS

2022 telecons, telecons, telecons…

- LWG review of P0009 wanted naming changes for problematic names
- `pointer` —> `data_handle_type`
  - Really is an opaque handle to data
  - Need not be dereferencable or indexable
  - Follows precedence of `std::thread::native_handle_type`
- Similar reasoning for `mdspan::data()` —> `mdspan::data_handle()`
- `contiguous` —> `exhaustive`
  - `contiguous` implies linear order, which isn't necessarily true

- Separate paper from P0009 to lessen risk of P0009 not making C++23
  - 13-13-0-0-0

P2613R1 ADD THE MISSING `EMPTY` TO `MDSPAN`

2022 telecons, telecons, telecons…

- Add empty() to go along with size()
- Separate paper from P0009 to lessen risk of P0009 not making C++23
  - Almost didn't make it, as P2613R0 had a wording bug
  - 10-10-1-1-0

```
[[nodiscard]] constexpr bool empty() const noexcept;
```

P0009R18 MDSPAN

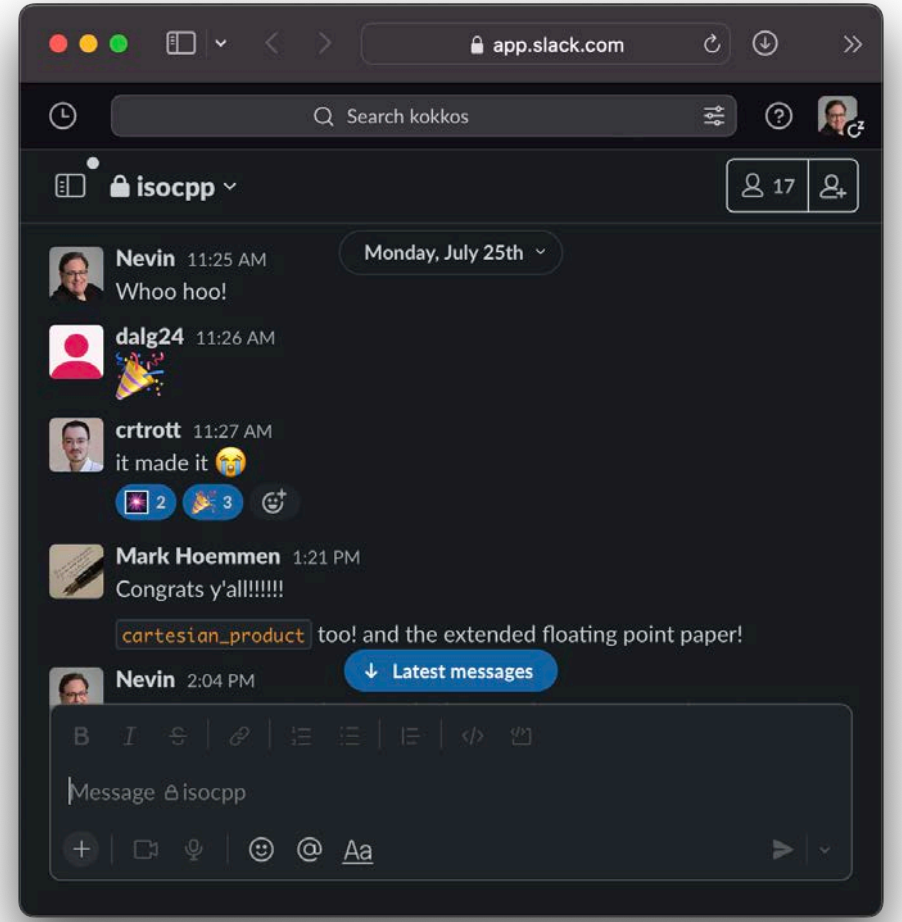
Christian Trott, D.S. Hollman, Damien Lebrun-Grandie, Mark Hoemmen, Daniel Sunderland, H. Carter Edwards, Bryce Adelstein Lelbach, Mauro Bianco, Ben Sander, Athanasios Iliopoulos, John Michopoulos, Nevin Liber

- Apply the changes in P0009R18 (MDSPAN) to the C++ working paper
  - ***Unanimous consent***
  - 2022-Jul-25 11:25 am CDT

BUT THAT WAS THE C++20 PRAGUE CELEBRATION

BUT THAT WAS THE C++20 PRAGUE CELEBRATION

HERE IS THE P0009 MDSPAN CELEBRATION:



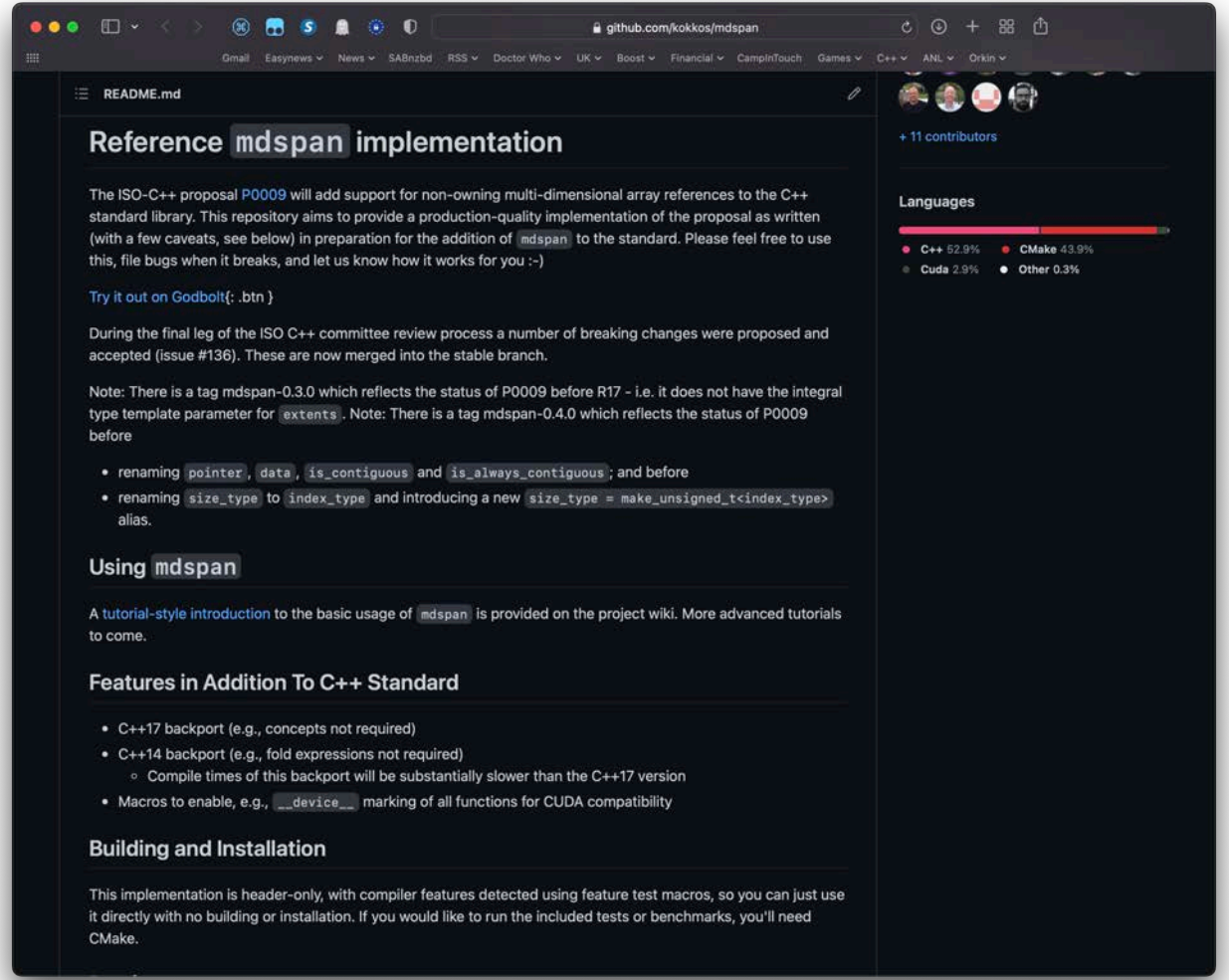
MDSPAN

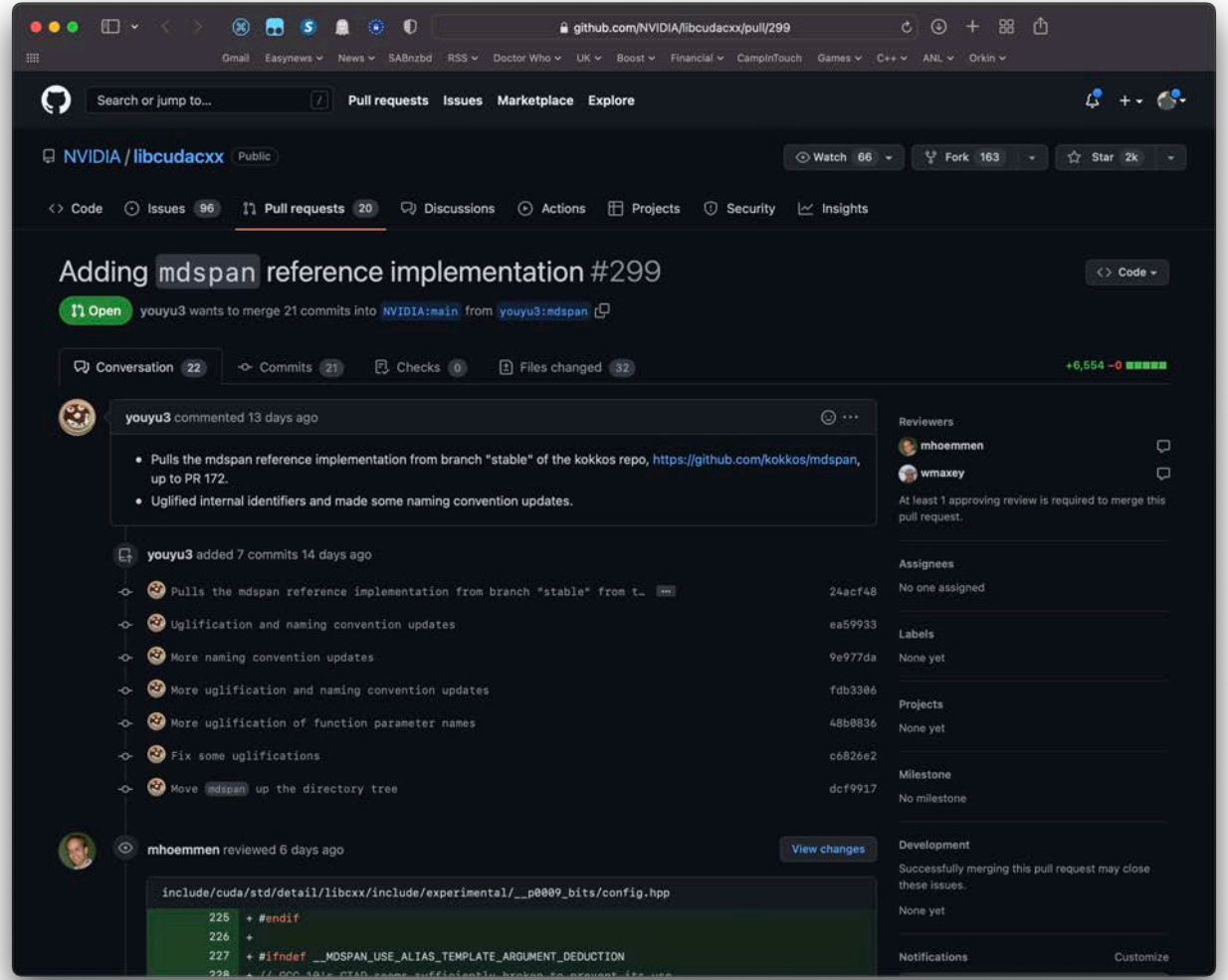
http://eel.is/c++draft/views#mdspan.syn

- As of 2022-August-17:

24.7.4 Header <mdspan> synopsis [mdspan.syn]

```
namespace std {
  // [mdspan.extents], class template extents
  template<class IndexType, size_t... Extents>
    class extents;

  // [mdspan.extents.dextents], alias template dextents
  template<class IndexType, size_t Rank>
    using dextents = see below;

  // [mdspan.layout], layout mapping
  struct layout_left;
  struct layout_right;
  struct layout_stride;

  // [mdspan.accessor.default], class template default_accessor
  template<class ElementType>
    class default_accessor;

  // [mdspan.mdspan], class template mdspan
  template<class ElementType, class Extents, class LayoutPolicy = layout_right,
           class AccessorPolicy = default_accessor<ElementType>>
    class mdspan;
}
```


github.com/kokkos/mdspan
README.md
Reference mdspan implementation
The ISO-C++ proposal P0009 will add support for non-owning multi-dimensional array references to the C++ standard library. This repository aims to provide a production-quality implementation of the proposal as written (with a few caveats, see below) in preparation for the addition of mdspan to the standard. Please feel free to use this, file bugs when it breaks, and let us know how it works for you :-)
Try it out on Godbolt{: .btn }
During the final leg of the ISO C++ committee review process a number of breaking changes were proposed and accepted (issue #136). These are now merged into the stable branch.
Note: There is a tag mdspan-0.3.0 which reflects the status of P0009 before R17 - i.e. it does not have the integral type template parameter for extents. Note: There is a tag mdspan-0.4.0 which reflects the status of P0009 before
renaming pointer, data, is_contiguous and is_always_contiguous; and before
renaming size_type to index_type and introducing a new size_type = make_unsigned_t<index_type> alias.
Using mdspan
A tutorial-style introduction to the basic usage of mdspan is provided on the project wiki. More advanced tutorials to come.
Features in Addition To C++ Standard
C++17 backport (e.g., concepts not required)
C++14 backport (e.g., fold expressions not required)
Compile times of this backport will be substantially slower than the C++17 version
Macros to enable, e.g., __device__ marking of all functions for CUDA compatibility
Building and Installation
This implementation is header-only, with compiler features detected using feature test macros, so you can just use it directly with no building or installation. If you would like to run the included tests or benchmarks, you'll need CMake.
+ 11 contributors
Languages
C++ 52.9%
CMake 43.9%
Cuda 2.9%
Other 0.3%


github.com/NVIDIA/libcudacxx/pull/299
Search or jump to...
Pull requests
Issues
Marketplace
Explore
NVIDIA / libcudacxx
Public
Watch 66
Fork 163
Star 2k
Code
Issues 96
Pull requests 20
Discussions
Actions
Projects
Security
Insights
Adding mdspan reference implementation #299
Code
Open
youyu3 wants to merge 21 commits into NVIDIA:main from youyu3:mdspan
Conversation 22
Commits 21
Checks 0
Files changed 32
+6,554 −0
youyu3 commented 13 days ago
Pulls the mdspan reference implementation from branch "stable" of the kokkos repo, https://github.com/kokkos/mdspan, up to PR 172.
Uglified internal identifiers and made some naming convention updates.
youyu3 added 7 commits 14 days ago
Pulls the mdspan reference implementation from branch "stable" from t… 24acf48
Uglification and naming convention updates ea59933
More naming convention updates 9e977da
More uglification and naming convention updates fdb3306
More uglification of function parameter names 48b0836
Fix some uglifications c6826e2
Move mdspan up the directory tree dcf9917
mhoemmen reviewed 6 days ago
View changes
include/cuda/std/detail/libcxx/include/experimental/__p0009_bits/config.hpp
225 + #endif
226 +
227 + #ifndef __MDSPAN_USE_ALIAS_TEMPLATE_ARGUMENT_DEDUCTION
Reviewers
mhoemmen
wmaxey
At least 1 approving review is required to merge this pull request.
Assignees
No one assigned
Labels
None yet
Projects
None yet
Milestone
No milestone
Development
Successfully merging this pull request may close these issues.
None yet
Notifications
Customize

SYCL & MDSPAN

SYCL & MDSPAN

Tom Deakin, Dániel Berényi, Nevin Liber
Ronan Keryell, Roland Schulz, Thomas Applencourt, James Brodman, Aksel Alpay, Gregory Lueck, Gordon Brown, Tadej Ciglarič

- A small subgroup of the Khronos SYCL Committee started thinking about and fleshing out how we can take advantage of `mdspan` in SYCL about a year ago
- First thoughts
  - Accessors
  - Unified Shared Memory (USM)
  - Strides, offsets and sub-buffers
  - More than three dimensions
  - Unify buffers and images

SYCL ACCESSORS

```
template <typename DataT, int Dimensions, access_mode AccessMode, target AccessTarget, ...>
class accessor { /* ... */ };
```

- `accessor` is the non-owning view of `sycl::buffer`
  - `access_mode`: `read`, `write`, `read_write`
  - `target`: `device`, `host_task`

- `mdspan` improvements over `accessors`
  - `LayoutPolicy` - flexibility for order data is stored
  - `AccessPolicy` - `restrict`, atomic, `volatile`, etc.

SYCL ACCESSORS

```
template <typename DataT, int Dimensions, access_mode AccessMode, target AccessTarget, ...>
class accessor { /* ... */ };
```

- `mdspan` improvements over accessors (continued)
  - Rectangular copies
    - USM copies from host
    - Async copy to/from all memory space combinations

- Alternative
  - Add these features to SYCL accessor

SYCL & MDSPAN

- `embedded_ptr` (hipSYCL)
  - Lightweight (compared with accessor) to get pointers to data inside kernels
  - In general we can't use raw pointers
    - Can't always share between host and device
  - We can use the `embedded_ptr` to directly create construct an `mdspan`

```
sycl::buffer<double, 2> A = {N,P};
embedded_ptr p_A {A, sycl::read_only};
cgh.parallel_for(...) {
    std::mdspan md_A {p_A, {N, P}};
}
```

SYCL & MDSPAN

Current Status

- Grew the subgroup to include implementers
- Working on proposal for next Khronos F2F Meeting (October 2022 - Phoenix)
  - Buffer accessor `mdspan`
  - `embedded_ptr` `mdspan`
  - USM `mdspan`
  - C++23 baseline?
    - `mdspan::operator[]` requires it

- ***Stay tuned!***

KOKKOS

KOKKOS

- Refactoring `View` to use `mdspan`
- Papers targeting C++26
  - P1673 A free function linear algebra interface based on the BLAS
  - P1684 `mdarray`
  - P2630 `submdspan`

P1684 MDARRAY

P1684R0 AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

mdarray

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class ContainerPolicy = see-below>
  class mdarray;
```

- Adaptor
  - `stack`, `queue`, `priority_queue`, `flat_set`, `flat_map`

P1684R0 AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

ContainerPolicy

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class ContainerPolicy = see-below>
  class mdarray;
```

- "Replaces" `AccessorPolicy` from `mdspan`
  - Generalization of needed contiguous container features
  - `create()` method

P1684R0 AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

Cologne 2019

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class ContainerPolicy = see-below>
  class mdarray;
```

- Polls
  - Do this as containers (md_array, md_vector?) instead of as adaptor? 0 7 2 2 3
  - Continue work and come back (we believe this is a problem the standard should solve? 8 8 3 2 0

P1684R0 AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

Container Adaptor

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class ContainerPolicy = see-below>
  class mdarray;
```

- Why an adaptor?
  - `array<T,N>`, `vector<T,A>`, `static_vector<T,N>`, `small_vector<T,N,A>`
  - Device specific containers (sometimes C++ Standard Library containers won't work)
  - User defined containers

P1684R1 AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class ContainerPolicy = see-below>
  class mdarray;
```

- Uses `Container`, not `ContainerPolicy`
  - Defaults to `array` when all static extents
  - Otherwise, defaults to `vector`
- Poll: The default container should be `std::vector`? 3 4 3 0 0

P1684R2 AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class ContainerPolicy = vector<ElementType>>
  class mdarray;
```

- Poll:
  - We support the presented container adapter design 8 8 0 1 0

P1684R3 AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class ContainerPolicy = vector<ElementType>>
  class mdarray;
```

- Consistent with C++23 `mdspan`
- Added size constructible container requirements

P1684R4? AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

Strong Invariants

- `mdarray` with all static extents is not default constructible
  - Move assignment modifies both the source and destination underlying containers
  - What is the moved-from state of an `mdarray` of all static extents?
    - Valid-but-unspecified state of the underlying container isn't sufficient
  - What is the state of both the source and destination if move assignment throws?
  - Common standard containers:
    - `array` - no problem (`array` elements in moved-from state won't break `mdarray` invariants)
    - `vector::clear()` isn't sufficient to maintain the invariant
    - User defined containers?
      - Even if `clear()` solved it (which is doesn't), in general `clear()` can throw too
- Other adaptors with invariants don't answer these questions either
  - Will probably file an NB comment for `flat_map` (, `flat_set`, `priority_queue`)

P2630 SUBMDSPAN

P2630 SUBMDSPAN

```
template<class T, class E, class L, class A,
         class... SliceArgs)
auto submdspan(mdspan<T,E,L,A> x, SliceArgs... args);
```

- Returns a different `mdspan` type
- Originally part of P0009
- Added customization points so that `submdspan` can work with user-defined policies
- Added the ability to specify slices as compile time values

P2630 SUBMDSPAN

SliceArgs

- `index_type` (from input `mdspan`)
  - Rank of resulting `mdspan` is one less than the input `mdspan`
  - Contains only elements where the index matches this slice specifier
- `tuple<index_type, index_type>`
  - Begin to end subrange of elements
- `full_extent_t`
  - Full range of indices
- `strided_index_range<OffsetType, ExtentType, StrideType>{.offset, .extent, .stride}`

- If any of `index_type`, `OffsetType`, `ExtentType`, `StrideType` is `integral_constant`
  - Compile time constant baked into the `mdspan` return type

P2630 SUBMDSPAN

`strided_index_range{.offset, .extent, .stride}`

- `offset`
  - The start index
- `extent`
  - Length of the subrange (*not* the end index)
- `stride`
  - Stride within that subrange

P2630 SUBMDSPAN

Customization Points

```
template<class Mapping, class ... SliceArgs>
auto submdspan_mapping(const Mapping&, SliceArgs...) { /* ... */ }

template<class Mapping, class ... SliceArgs>
size_t submdspan_offset(const Mapping&, SliceArgs...) { /* ... */ }


template<class T, class E, class L, class A,
         class ... SliceArgs)
auto submdspan(const mdspan<T,E,L,A>& src, SliceArgs ... args) {
  size_t sub_offset = submdspan_offset(src.mapping(), args...); // ADL
  auto sub_map = submdspan_mapping(src.mapping(), args...);     // ADL
  typename A::offset_policy sub_acc(src.accessor());
  typename A::offset_policy::data_handle_type
    sub_handle = src.accessor().offset(src.data_handle(), sub_offset);
  return mdspan(sub_handle, sub_map, sub_acc); // Customizations
}
.
```

SPECIAL THANKS:

ARGONNE NATIONAL LABORATORY

KOKKOS TEAM

C++ COMMITTEE

KHRONOS SYCL COMMITTEE

…AND A CAST OF TENS? HUNDREDS?

(THANK THEM / BLAME ME)

- This was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative. Additionally, this research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

Q&A