`mdspan::size_type` should be `index_type`

Document number: P2599R1
Date: 2022-06-14
Project: Programming Language C++, Library Evolution Working Group
Reply-to: Nevin “☺” Liber, nliber@anl.gov

Table of Contents

Revisions

R1

In order to strengthen consensus, LEWG requested that in addition to the changes requested in P2599R0 (change all current references of `size_type` to `index_type`), we also add a new `size_type` typedef mapped to the unsigned type corresponding to what would now be `index_type`.

Polls

__POLL: 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.__

Strongly Favor	Weakly Favor	Neutral	Weakly Against	Strongly Against
2	7	2	2	1

__Attendance:__ 21

__# of Authors:__ 1

__Author Position:__ SF

__Outcome:__ Weak consensus in favor.

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.

__POLL: Rename `mdspan` and friend's `size_type` member to `index_type` and have a `size_type` member be present only if `index_type` is unsigned.__

Author note: not polled, as the poll below had consensus.

__POLL: `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>`).__

Strongly Favor	Weakly Favor	Neutral	Weakly Against	Strongly Against
3	9	1	1	0

__Attendance:__ 19

__# of Authors:__ 1

__Author Position:__ SF

__Outcome:__ Consensus in favor, and stronger consensus that the paper as written.

WA: It's additional complexity.

Introduction

With the adoption of P2553R1, `mdspan::size_type` may now be a signed type. `size_type` is no longer an appropriate name for this type and it should be changed to `index_type`.

Motivation and Scope

Throughout the C++ standard, `size_type` stands for an unsigned type. `mdspan` and its related class templates should be consistent with this.

When P2553R0 was proposed, `extents::size_type` was going to be constrained to `unsigned_integral`. At the request of LEWG, that constraint was removed in P2553R1 and adopted via electronic polling.

Now that it can be a signed type, `size_type` is no longer the correct name for this. It should revert back to `index_type`, which was used in `mdspan` until P0009R11 when the following change was made:

> Change all the sizes
> from `ptrdiff_t` to `size_t` and `index_type` to `size_type`, for consistency with `span` and the rest of the standard library

In addition to `extents`, there are other class templates which take `Extents` as a template parameter and adopt the `size_type` typedef from `Extents` into their interface. Those class templates should also have their `size_type` typedefs changed to `index_type`.

LEWG requested that a new `size_type` that corresponds to the unsigned version of `index_type` also be added to these class templates.

Specifically, the following class templates should replace their usage of `size_type` with `index_type` and then add a new `size_type`:

- `extents`
- `layout_left::mapping`
- `layout_right::mapping`
- `layout_stride::mapping`
- `mdspan`

Impact On the Standard

Given that `mdspan` and its related classes are new class templates for C++23, the impact should be minimal. Also, no feature test macro should be necessary.

Technical Specifications

The changes proposed here are:

- Normatively change the spelling of `size_type` to `index_type`
- Editorially change the spelling of template parameter `SizeT` / `SizeType` to `IndexType`
- Editorially change the spelling of template parameter `OtherSizeT` / `OtherSizeType` to `OtherIndexType`
- Editorially change the spelling of template parameter `SizeTypes` to `IndexTypes`
-

Then apply the following additions:

- To `extents`, normatively add the public definition `using size_type = make_unsigned_t<index_type>;`
- To `layout_left::mapping`, `layout_right::mapping`, `layout::stride::mapping` and `mdspan`, add the public definition `using size_type = typename extents_type::size_type;`

Note: P0009 and P2553 are currently undergoing revisions as requested by LWG. If this proposal is approved, the author will rebase these changes on the result of those changes.

Acknowledgements

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.

References

P0009 mdspan, Christian Trott *et al.*
P2553 Make mdspan size_type controllable, Christian Trott *et al.*

Appendix

Rendered diffs against the 9e0e246 source commit of P0009 and P2553 (found at commit 9980c74).

```
template<size_t ... Extents>
using extents = basic_extents<size_t, Extents...>;
```

LEWG would need to decide whether to make dextents have a `size_type` template parameter or not.

2.3 Why we can't fix this later

In principle we could add a second extents type later, though it may break code such as the one shown before (in the sense that it wouldn't generally work for every instance of `mdspan` anymore):

```
template<class T, class L, class A, size_t ... Extents>
void foo(mdspan<T,extents<Extents...>,L,A> a) {...}
```

3 Editing Notes

The proposed changes are relative to the working draft of the standard as of P0009R16.

4 Wording

4.1 In 22.7.X [mdspan.syn]

Replace:

```
template<size_t... Extents>
  class extents;

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

with:

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

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

4.2 In 22.7.X.1 [mdspan.extents.overview]:

— In the synopsis replace:

```
template<size_t... Extents>
class extents {
public:
  using size_type = size_t;
  using rank_type = size_t;
```

with

```
template<class IndexType, size_t... Extents>
class extents {
public:
  using index_type = IndexType;
  using size_type = make_unsigned_t<index_type>;
  using rank_type = size_t;
```

— In the synopsis replace:

```
static constexpr size_type static_extent(rank_type) noexcept;
```

with

```
static constexpr size_t static_extent(rank_type) noexcept;
```

— In the synopsis replace:

```
template<size_t... OtherExtents>
  explicit(see below)
  constexpr extents(const extents<OtherExtents...>&) noexcept;
```

with:

```
template<class OtherIndexType, size_t... OtherExtents>
  explicit(see below)
  constexpr extents(const extents<OtherIndexType, OtherExtents...>&) noexcept;
```

— In the synopsis replace:

```
// [mdspan.extents.cmp], extents comparison operators
template<size_t... OtherExtents>
  friend constexpr bool operator==(const extents&, const extents<OtherExtents...>&) noexcept;
```

with:

```
// [mdspan.extents.cmp], extents comparison operators
template<class OtherIndexType, size_t... OtherExtents>
  friend constexpr bool operator==(const extents&, const extents<OtherIndexType, OtherExtents...>&) noe
```

— Before paragraph 2 of [mdspan.extents.overview] insert a new paragraph 2:

Mandates:

— `IndexType` is an integral type other than `bool`, and

— each element of `Extents` is either equal to `dynamic_extent`, or is a representable value of type `IndexType`.

4.3 In subsection 22.7.X.3 [mdspan.extents.ctor]

— Change the following:

```
template<size_t... OtherExtents>
  explicit((((Extents!=dynamic_extent) && (OtherExtents==dynamic_extent)) || ... ))
  constexpr extents(const extents<OtherExtents...>& other) noexcept;
```

to:

```
template<class OtherIndexType, size_t... OtherExtents>
  explicit(see below)
  constexpr extents(const extents<OtherIndexType, OtherExtents...>& other) noexcept;
```

— Change Paragraph 2 (*Preconditions* element of the above constructor) to:

Preconditions:

— `other.extent(r)` equals E_r for each r for which E_r is a static extent, and

— either

— `sizeof...(OtherExtents)` is zero, or

— `other.extent(r)` is a representable value of type `index_type` for all rank index `r` of `other`.

— Add new paragraph 4 after paragraph 3 (*Postconditions* element of the above constructor):

Remarks: The expression inside `explicit` is equivalent to:

```
(((Extents!=dynamic_extent) && (OtherExtents==dynamic_extent)) || ... ) ||
(numeric_limits<index_type>::max() < numeric_limits<OtherIndexType>::max())
```

— Change (the old) Paragraph 6 (*Preconditions* element of the `extents(IndexTypes...)` constructor) to:

Preconditions:

— If `sizeof...(IndexTypes) != rank_dynamic()` is `true`, `exts_arr[r]` equals E_r for each r for which E_r is a static extent, and

— either

— `true` if `sizeof...(exts) == 0` is `true`, or

— each element of `exts` is nonnegative and is a representable value of type `index_type`.

— Change Paragraph 9 (*Preconditions* element of the `extents` constructors that take `span` resp. `const array&`) to:

Preconditions:

— If `N != rank_dynamic()` is `true`, `exts[r]` equals E_r for each r for which E_r is a static extent, and

— either

— `N` is zero, or

— for all `r` in the range of [0,N), `exts[r]` is nonnegative and is a representable value of type `index_type`.

— Change paragraph 12 (*Remarks* element of `extents(Integrals...)` deduction guide) to:

Remarks: The deduced type is `dextents<size_t, sizeof...(Integrals)>`.

4.4 In subsection 22.7.X.4 [mdspan.extents.obs]

— Replace the following:

```
static constexpr size_type static_extent(rank_type i) const noexcept;
```

with:

```
static constexpr size_t static_extent(rank_type i) const noexcept;
```

4.5 In subsection 22.7.X.5 [mdspan.extents.cmp]

— Replace the following:

```
template<size_t... OtherExtents>
  friend constexpr bool operator==(const extents& lhs,
                                   const extents<OtherExtents...>& rhs) noexcept;
```

with:

```
template<class OtherIndexType, size_t... OtherExtents>
  friend constexpr bool operator==(const extents& lhs,
                                   const extents<OtherIndexType, OtherExtents...>& rhs) noexcept;
```

4.6 In subsection 22.7.X.6 [mdspan.extents.dextents]

— Replace section with:

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

1 *Result:* A type `E` that is a specialization of `extents` such that `E::rank() == Rank && E::rank() == E::rank_dynamic()` is `true` and `E::index_type` denotes `IndexType`.

4.7 In subsection 22.7.X.3.1 [mdspan.layoutleft.ctor]

— After paragraph 3 (*Constraints* element of `mapping(const mapping<OtherExtents>& other)` constructor) add new paragraph:

Preconditions: `other.required_span_size()` is a representable value of type `index_type` ([basic.fundamental]).

— After paragraph 5 (*Constraints* element of constructor converting from `layout_right::mapping`) add new paragraph:

Preconditions: `other.required_span_size()` is a representable value of type `index_type` ([basic.fundamental]).

— Change paragraph 8 (*Preconditions* element of constructor converting from `layout_stride::mapping`) to:

Preconditions:

— If `extents_type::rank() > 0` is `true`, then for all r in the range [`0`, `extents_type::rank()`), `other.stride(r)` equals `extents().`*fwd-prod-of-extents*`(r)`, and

— `other.required_span_size()` is a representable value of type `index_type` ([basic.fundamental]).

4.8 In subsection 22.7.X.4.1 [mdspan.layoutright.ctor]

— After paragraph 3 (*Constraints* element of `mapping(const mapping<OtherExtents>& other)` constructor) add new paragraph:

Preconditions: `other.required_span_size()` is a representable value of type `index_type` ([basic.fundamental]).

— After paragraph 5 (*Constraints* element of constructor converting from `layout_left::mapping`) add new paragraph:

Preconditions: `other.required_span_size()` is a representable value of type `index_type` ([basic.fundamental]).

— Change paragraph 8 (*Preconditions* element of constructor converting from `layout_stride::mapping`) to:

Preconditions:

— If `extents_type::rank() > 0` is `true`, then for all r in the range [`0`, `extents_type::rank()`), `other.stride(r)` equals `extents().`*rev-prod-of-extents*`(r)`, and

— `other.required_span_size()` is a representable value of type `index_type` ([basic.fundamental]).

4.9 In subsection 22.7.X.1 [mdspan.mdspan.overview]

— In the synopsis replace:

```
static constexpr size_type static_extent(size_t r) { return Extents::static_extent(r); }
```

with:

```
static constexpr size_t static_extent(size_t r) { return Extents::static_extent(r); }
```

— In the synopsis replace:

```
    constexpr size_type size() const;
```

with

```
constexpr size_t size() const;
```

— In the synopsis replace:

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

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

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

with:

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

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

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

5 Implementation

There is an mdspan implementation available at https://github.com/kokkos/mdspan/.

6 Related Work

— P0009 : `mdspan`

18 *Returns:* `true` only if for every rank index r of `m.extents()` there exists a nonnegative integer s_r such that, for all `i` where `(i+`d_r`)` is a multidimensional index in `m.extents()` ([mdspan.terms]), `m((i+`d_r`)...) - m(i...)` equals s_r. [Note: This implies that for a strided layout `m(`$i_0, ..., i_k$`) = m(0, ..., 0) +`$i_0 * s_0 + ... + i_k * s_k$. — end note] [Note: A mapping may return `false` even if the condition is met. For certain layouts it may not be feasible to determine efficiently whether the layout is strided.— end note]

```
m.stride(r)
```

19 *Preconditions:* `m.is_strided()` is `true`.

20 *Result:* `typename M::index_type`

21 *Returns:* s_r (a nonnegative integer) as defined in `m.is_strided()` above.

```
M::is_always_unique()
```

22 *Result:* A constant expression ([expr.const]) of type `bool`.

23 *Returns:* `true` only if `m.is_unique()` is `true` for all possible objects `m` of type `M`. [Note: A mapping may return `false` even if the above condition is met. For certain layout mappings it may not be feasible to determine whether every instance is unique.— end note]

```
M::is_always_contiguous()
```

24 *Result:* A constant expression ([expr.const]) of type `bool`.

25 *Returns:* `true` only if `m.is_contiguous()` is `true` for all possible objects `m` of type `M`. [Note: A mapping may return `false` even if the above condition is met. For certain layout mappings it may not be feasible to determine whether every instance is contiguous.— end note]

```
M::is_always_strided()
```

26 *Result:* A constant expression ([expr.const]) of type `bool`.

27 *Returns:* `true` only if `m.is_strided()` is `true` for all possible objects `m` of type `M`. [Note: A mapping may return `false` even if the above condition is met. For certain layout mappings it may not be feasible to determine whether every instance is strided.— end note]

24.7. .3 Layout mapping policy requirements [mdspan.layoutpolicy.reqmts]

1 A type `MP` meets the *layout mapping policy* requirements if for a type `E` that is a specialization of `extents`, `MP::mapping<E>` is valid and denotes a type `X` that meets the layout mapping requirements ([mdspan.layout.reqmts]), and for which the *qualified-id* `X::layout_type` is valid and denotes the type `MP` and the *qualified-id* `X::extents_type` denotes `E`.

24.7. .4 Layout mapping policies [mdspan.layoutpolicy.overview]

```
namespace std {

struct layout_left {
  template<class Extents>
    class mapping;
};
struct layout_right {
  template<class Extents>
    class mapping;
};
struct layout_stride {
  template<class Extents>
    class mapping;
};
```

```
}
```

1 Each of `layout_left`, `layout_right`, and `layout_stride` meets the layout mapping policy requirements and is a trivial type.

24.7. .5 Class template `layout_left::mapping` [mdspan.layoutleft]

24.7. .5.1 Overview [mdspan.layoutleft.overview]

1 `layout_left` provides a layout mapping where the leftmost extent has stride 1, and strides increase left-to-right as the product of extents.

```
namespace std {

template<class Extents>
class layout_left::mapping {
  public:
    using extents_type = Extents;
    using index_type = typename extents_type::index_type;
    using size_type = typename extents_type::size_type;
    using rank_type = typename extents_type::rank_type;
    using layout_type = layout_left;

    // [mdspan.layoutleft.ctor], Constructors
    constexpr mapping() noexcept = default;
    constexpr mapping(const mapping&) noexcept = default;
    constexpr mapping(const extents_type&) noexcept;
    template<class OtherExtents>
      explicit(!is_convertible_v<OtherExtents, extents_type>)
      constexpr mapping(const mapping<OtherExtents>&) noexcept;
    template<class OtherExtents>
      explicit(see below)
      constexpr mapping(const layout_right::mapping<OtherExtents>&) noexcept;
    template<class OtherExtents>
      explicit(extents_type::rank() > 0)
      constexpr mapping(const layout_stride::mapping<OtherExtents>&);

    constexpr mapping& operator=(const mapping&) noexcept = default;

    // [mdspan.layoutleft.obs], Observers
    constexpr const extents_type& extents() const noexcept { return extents_; }

    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 { return true; }
    static constexpr bool is_always_contiguous() noexcept { return true; }
    static constexpr bool is_always_strided() noexcept { return true; }

    static constexpr bool is_unique() noexcept { return true; }
    static constexpr bool is_contiguous() noexcept { return true; }
    static constexpr bool is_strided() noexcept { return true; }
```

```
    constexpr index_type stride(rank_type) const noexcept;

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

  private:
    extents_type extents_{}; // exposition only
};

}
```

2 If `Extents` is not a specialization of `extents`, then the program is ill-formed.

3 `layout_left::mapping<E>` is a trivially copyable type that models `regular` for each `E`.

24.7. .5.2 Constructors [mdspan.layoutleft.ctor]

```
constexpr mapping(const extents_type& e) noexcept;
```

1 *Preconditions:* The size of the multidimensional index space `e` is a representable value of type `index_type` ([basic.fundamental]).

2 *Effects:* Direct-non-list-initializes *extents_* with `e`.

```
template<class OtherExtents>
  explicit(!is_convertible_v<OtherExtents, extents_type>)
  constexpr mapping(const mapping<OtherExtents>& other) noexcept;
```

3 *Constraints:* `is_constructible_v<extents_type, OtherExtents>` is `true`.

4 *Effects:* Direct-non-list-initializes *extents_* with `other.extents()`.

```
template<class OtherExents>
  explicit(!is_convertible_v<OtherExtents, extents_type>)
  constexpr mapping(const layout_right::mapping<OtherExtents>& other) noexcept;
```

5 *Constraints:*

(5.1) — `extents_type::rank() <= 1` is `true`, and

(5.2) — `is_constructible_v<extents_type, OtherExtents>` is `true`.

6 *Effects:* Direct-non-list-initializes *extents_* with `other.extents()`.

```
template<class OtherExtents>
  explicit(extents_type::rank() > 0)
  constexpr mapping(const layout_stride::mapping<OtherExtents>& other);
```

7 *Constraints:* `is_constructible_v<extents_type, OtherExtents>` is `true`.

8 *Preconditions:* If `extents_type::rank() > 0` is `true`, then for all r in the range [0, `extents_type::rank()`), `other.stride(`r`)` equals `extents().`*`fwd-prod-of-extents`*`(`r`)`.

9 *Effects:* Direct-non-list-initializes *extents_* with `other.extents()`.

24.7. .5.3 Observers [mdspan.layoutleft.obs]

```
constexpr index_type required_span_size() const noexcept;
```

1 *Returns:* `extents().`*`fwd-prod-of-extents`*`(extents_type::rank())`.

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

2 *Constraints:*

(2.1) — `sizeof...(Indices) == extents_type::rank()` is `true`,

(2.2) — `(is_convertible_v<Indices, index_type> && ...)` is `true`, and

(2.3) — `(is_nothrow_constructible_v<index_type, Indices> && ...)` is `true`.

3 *Preconditions:* `static_cast<index_type>(i)` is a multidimensional index in *extents*_ ([mdspan.terms]).

4 *Effects:* Let `P` be a parameter pack such that `is_same_v<index_sequence_for<Indices...>, index_sequence<P...>>` is `true`. Equivalent to: `return ((static_cast<index_type>(i)*stride(P)) + ... + 0);`

```
constexpr index_type stride(rank_type i) const;
```

5 *Constraints:* `extents_type::rank() > 0` is `true`.

6 *Preconditions:* `i < extents_type::rank()` is `true`.

7 *Returns:* `extents().`*`fwd-prod-of-extents`*`(i)`.

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

8 *Constraints:* `extents_type::rank() == OtherExtents::rank()` is `true`.

9 *Effects:* Equivalent to: `return x.extents() == y.extents();`

24.7. .6 Class template `layout_right::mapping` [mdspan.layoutright]

24.7. .6.1 Overview [mdspan.layoutright.overview]

1 `layout_right` provides a layout mapping where the rightmost extent is stride 1, and strides increase right-to-left as the product of extents.

```
namespace std {

template<class Extents>
class layout_right::mapping {
  public:
    using extents_type = Extents;
    using index_type = typename extents_type::index_type;
    using size_type = typename extents_type::size_type;
    using rank_type = typename extents_type::rank_type;
    using layout_type = layout_right;

    // [mdspan.layoutright.ctor], Constructors
    constexpr mapping() noexcept = default;
    constexpr mapping(const mapping&) noexcept = default;
    constexpr mapping(const extents_type&) noexcept;
    template<class OtherExtents>
      explicit(!is_convertible_v<OtherExtents, extents_type>)
      constexpr mapping(const mapping<OtherExtents>&) noexcept;
    template<class OtherExtents>
      explicit(see below)
      constexpr mapping(const layout_left::mapping<OtherExtents>&) noexcept;
    template<class OtherExtents>
      explicit(extents_type::rank() > 0)
      constexpr mapping(const layout_stride::mapping<OtherExtents>&) noexcept;

    constexpr mapping& operator=(const mapping&) noexcept = default;
```

```
    // [mdspan.layoutright.obs], Observers
    constexpr const extents_type& extents() const noexcept { return extents_; }

    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 { return true; }
    static constexpr bool is_always_contiguous() noexcept { return true; }
    static constexpr bool is_always_strided() noexcept { return true; }

    static constexpr bool is_unique() noexcept { return true; }
    static constexpr bool is_contiguous() noexcept { return true; }
    static constexpr bool is_strided() noexcept { return true; }

    constexpr index_type stride(rank_type) const noexcept;

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

  private:
    extents_type extents_{}; // exposition only
};
}
```

2 If `Extents` is not a specialization of `extents`, then the program is ill-formed.

3 `layout_right::mapping<E>` is a trivially copyable type that models `regular` for each `E`.

24.7. .6.2 Constructors [mdspan.layoutright.ctor]

```
constexpr mapping(const extents_type& e) noexcept;
```

1 *Preconditions:* The size of the multidimensional index space `e` is a representable value of type `index_type` ([basic.fundamental]).

2 *Effects:* Direct-non-list-initializes *extents_* with `e`.

```
template<class OtherExtents>
  explicit(!is_convertible_v<OtherExtents, extents_type>)
  constexpr mapping(const mapping<OtherExtents>& other) noexcept;
```

3 *Constraints:* `is_constructible_v<extents_type, OtherExtents>` is `true`.

4 *Effects:* Direct-non-list-initializes *extents_* with `other.extents()`.

```
template<class OtherExtents>
  explicit(!is_convertible_v<OtherExtents, extents_type>)
  constexpr mapping(const layout_left::mapping<OtherExtents>& other) noexcept;
```

5 *Constraints:*

(5.1) — `extents_type::rank() <= 1` is `true`, and

(5.2) — `is_constructible_v<extents_type, OtherExtents>` is `true`.

6 *Effects:* Direct-non-list-initializes *extents_* with `other.extents()`.

```
template<class OtherExtents>
  explicit(extents_type::rank() > 0)
  constexpr mapping(const layout_stride::mapping<OtherExtents>& other) noexcept;
```

Constraints: `is_constructible_v<extents_type, OtherExtents>` is `true`.

Preconditions: If `extents_type::rank() > 0` is `true`, then for all r in the range [0, `extents_type::rank()`), `other.stride(`r`)` equals `extents().`*`rev-prod-of-extents`*`(`r`)`.

Effects: Direct-non-list-initializes *extents_* with `other.extents()`.

24.7. .6.3 Observers [mdspan.layoutright.obs]

```
index_type required_span_size() const noexcept;
```

Returns: `extents().`*`fwd-prod-of-extents`*`(extents_type::rank())`

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

Constraints:

(2.1) — `sizeof...(Indices) == extents_type::rank()` is `true`, and

(2.2) — `(is_convertible_v<Indices, index_type> && ...)` is `true`.

(2.3) — `(is_nothrow_constructible_v<index_type, Indices> && ...)` is `true`.

Preconditions: `static_cast<index_type>(i)` is a multidimensional index in *extents_* ([mdspan.terms]).

Effects: Let `P` be a parameter pack such that `is_same_v<index_sequence_for<Indices...>, index_sequence<P...>>` is `true`. Equivalent to: `return ((static_cast<index_type>(i)*stride(P)) + ... + 0);`

```
constexpr index_type stride(rank_type i) const noexcept;
```

Constraints: `extents_type::rank() > 0` is `true`.

Preconditions: `i < extents_type::rank()` is `true`.

Returns: `extents().`*`rev-prod-of-extents`*`(i)`

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

Constraints: `extents_type::rank() == OtherExtents::rank()` is `true`.

Effects: Equivalent to: `return x.extents() == y.extents();`

24.7. .7 Class template `layout_stride::mapping` [mdspan.layoutstride]

24.7. .7.1 Overview [mdspan.layoutstride.overview]

`layout_stride` provides a layout mapping where the strides are user-defined.

```
namespace std {

template<class Extents>
class layout_stride::mapping {
  public:
    using extents_type = Extents;
    using index_type = typename extents_type::index_type;
    using size_type = typename extents_type::size_type;
    using rank_type = typename extents_type::rank_type;
    using layout_type = layout_stride;
```

```
  private:
    static constexpr rank_type rank_ = extents_type::rank(); // exposition only

  public:
    // [mdspan.layoutstride.ctor], Constructors
    constexpr mapping() noexcept = default;
    constexpr mapping(const mapping&) noexcept = default;
    template<class IndexType>
    constexpr mapping(const extents_type&,
                      span<IndexType, rank_>) noexcept;
    template<class IndexType>
    constexpr mapping(const extents_type&,
                      const array<IndexType, rank_>&) noexcept;

    template<class StridedLayoutMapping>
      explicit(see below)
      constexpr mapping(const StridedLayoutMapping&) noexcept;

    constexpr mapping& operator=(const mapping&) noexcept = default;

    // [mdspan.layoutstride.obs], Observers
    constexpr const extents_type& extents() const noexcept { return extents_; }
    constexpr span<const index_type, rank_> strides() const noexcept
    { return strides_; }

    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 { return true; }
    static constexpr bool is_always_contiguous() noexcept { return false; }
    static constexpr bool is_always_strided() noexcept { return true; }

    static constexpr bool is_unique() noexcept { return true; }
    constexpr bool is_contiguous() const noexcept;
    static constexpr bool is_strided() noexcept { return true; }

    constexpr index_type stride(rank_type i) const noexcept { return strides_[i]; }

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

  private:
    extents_type extents_{}; // exposition only
    array<index_type, rank_> strides_{}; // exposition only
};
}
```

[2] If `Extents` is not a specialization of `extents`, then the program is ill-formed.

[3] `layout_stride::mapping<E>` is a trivially copyable type that models `regular` for each `E`.

24.7. .7.2 Exposition-only helpers [mdspan.layoutstride.expo]

1 Let *REQUIRED-SPAN-SIZE*(e, strides) be:

(1.1) — 1, if e.rank() == 0 is true, otherwise

(1.2) — 0, if the size of the multidimensional index space e is 0, otherwise

(1.3) — 1 plus the sum of products of (e.extent(r) - 1) and (strides[r]) for all r in the range [0, e.rank()).

2 Let *OFFSET*(m) be:

(2.1) — m(), if e.rank() == 0 is true, otherwise

(2.2) — 0, if the size of the multidimensional index space e is 0, otherwise

(2.3) — m(z...) for a pack of integers z that is a multidimensional index into m.extents() and each element of z equals 0.

3 Let *is-extents* be the exposition only variable template:

```
template<class T> constexpr bool is-extents = false;

template<size_t ... args> constexpr bool is-extents<extents<args...>> = true;
```

4 Let *layout-mapping-alike* be the exposition-only concept defined as follows:

```
template<class M>
concept layout-mapping-alike = requires {
  requires is-extents<typename M::extents_type>;
  { M::is_always_strided() } -> same_as<bool>;
  { M::is_always_contiguous() } -> same_as<bool>;
  { M::is_always_unique() } -> same_as<bool>;
  bool_constant<M::is_always_strided()>::value;
  bool_constant<M::is_always_contiguous()>::value;
  bool_constant<M::is_always_unique()>::value;
};
```

[Note: This concept checks that the functions M::is_always_strided(), M::is_always_contiguous(), and M::is_always_unique() exist, are constant expressions, and have a return type of bool. - end note]

24.7. .7.3 Constructors [mdspan.layoutstride.ctor]

```
template<class IndexType>
constexpr mapping(const extents_type& e, span<IndexType, rank_> s) noexcept;
template<class IndexType>
constexpr mapping(const extents_type& e, const array<IndexType, rank_>& s) noexcept;
```

1 *Constraints:*

(1.1) — is_convertible_v<const IndexType&, index_type> is true,

(1.2) — is_nothrow_constructible_v<index_type, const IndexType&> is true.

2 *Preconditions:*

(2.1) — s[i] > 0 is true for all i in the range [0, *rank_*).

(2.2) — *REQUIRED-SPAN-SIZE*(e, s) is a representable value of type index_type ([basic.fundamental]).

(2.3) — If *rank_* is greater than 0, then there exists a permutation P of the integers in the range [0, *rank_*), such that s[p_i] >= s[p_{i-1}] * e.extent(p_{i-1}) is true for all i in the range [1, *rank_*), where p_i is the i^{th} element of P. [Note: For layout_stride, this condition is necessary and sufficient for is_unique() to be true. — end note]

3 *Effects:* Direct-non-list-initializes *extents_* with e, and for all d in the range [0, *rank_*), direct-non-list-initializes *strides_*[d] with as_const(s[d]).

```
template<class StridedLayoutMapping>
  explicit(see below)
  constexpr mapping(const StridedLayoutMapping& other) noexcept;
```

4 *Constraints:*

(4.1) — *layout-mapping-alike*<StridedLayoutMapping> is satisfied.

(4.2) — `is_constructible_v<extents_type, typename StridedLayoutMapping::extents_type>` is `true`.

(4.3) — `StridedLayoutMapping::is_always_unique()` is `true`.

(4.4) — `StridedLayoutMapping::is_always_strided()` is `true`.

5 *Preconditions:*

(5.1) — `StridedLayoutMapping` meets the layout mapping requirements,

(5.2) — `other.stride(r) > 0` is `true` for all rank index `r` of `extents()`,

(5.3) — `other.required_span_size()` is a representable value of type `index_type` ([basic.fundamental]), and

(5.4) — *OFFSET*`(other) == 0` is `true`.

6 *Effects:* Direct-non-list-initializes *extents_* with `other.extents()`, and for all d in the range [0, *rank_*), direct-non-list-initializes *strides_*`[`d`]` with `other.stride(`d`)`.

7 *Remarks:* The expression inside `explicit` is equivalent to:

```
!(is_convertible_v<typename StridedLayoutMapping::extents_type, extents_type> && (
  is-mapping-of<layout_left, LayoutStrideMapping> ||
  is-mapping-of<layout_right, LayoutStrideMapping> ||
  is-mapping-of<layout_stride, LayoutStrideMapping>))
```

24.7. .7.4 Observers [mdspan.layoutstride.obs]

```
constexpr index_type required_span_size() const noexcept;
```

1 *Returns:* *REQUIRED-SPAN-SIZE*`(extents(),`*strides_*`)`.

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

2 *Constraints:*

(2.1) — `sizeof...(Indices) ==`*rank_* is `true`, and

(2.2) — `(is_convertible_v<Indices, index_type> && ...)` is `true`.

(2.3) — `(is_nothrow_constructible_v<index_type, Indices> && ...)` is `true`.

3 *Preconditions:* `static_cast<index_type>(i)` is a multidimensional index in *extents_* ([mdspan.terms]).

4 *Effects:* Let `P` be a parameter pack such that `is_same_v<index_sequence_for<Indices...>, index_sequence<P...>>` is `true`. Equivalent to: `return ((static_cast<index_type>(i)*stride(P)) + ... + 0);`

```
constexpr bool is_contiguous() const noexcept;
```

5 *Returns:*

(5.1) — `true` if *rank_* is 0.

(5.2) — Otherwise, `true` if there is a permutation P of the integers in the range [0, *rank_*), such that `stride(` p_0 `)` equals `1`, and `stride(` p_i `)` equals `stride(` p_{i-1} `) * extents().extent(` p_{i-1} `)` for i in the range [1, *rank_*), where p_i is the i^{th} element of P.

(5.3) — Otherwise, `false`.

```
template<class OtherMapping>
  friend constexpr bool operator==(const mapping& x, const OtherMapping& y) noexcept;
```

6 *Constraints:*

(6.1) — *`layout-mapping-alike`*`<OtherMapping>` is satisfied.

(6.2) — *`rank_`* `== OtherMapping::extents_type::rank()` is `true`.

(6.3) — `OtherMapping::is_always_strided()` is `true`.

7 *Preconditions:* `OtherMapping` meets layout mapping requirements.

8 *Returns:* `true` if `x.extents() == y.extents()` is `true`, *`OFFSET`*`(y) == 0` is `true`, and each of `x.stride(`r`) == y.stride(`r`)` is `true` for r in the range of [0, `x.extents.rank()`). Otherwise, `false`.

24.7. Accessor Policy [mdspan.accessor]

24.7. .1 General [mdspan.accessor.general]

1 An *accessor policy* defines types and operations by which a reference to a single object is created from an abstract data handle to a number of such objects and an index.

2 A range of indices $[0, N)$ is an *accessible range* of a given data handle and an accessor, if for each `i` in the range the accessor policy's `access` function produces a valid reference to an object.

3 In subclause 24.7. .2,

(3.1) — `A` denotes an accessor policy.

(3.2) — `a` denotes a value of type `A` or `const A`.

(3.3) — `p` denotes a value of type `A::pointer` or `const A::pointer`. [Note: The type `A::pointer` need not be dereferenceable. - end note]

(3.4) — `n`, `i` and `j` each denote values of type `size_t`.

24.7. .2 Requirements [mdspan.accessor.reqmts]

1 A type `A` meets the accessor policy requirements if

(1.1) — `A` models `copyable`,

(1.2) — `is_nothrow_move_constructible_v<A>` is `true`,

(1.3) — `is_nothrow_move_assignable_v<A>` is `true`,

(1.4) — `is_nothrow_swappable_v<A>` is `true`, and

(1.5) — the following types and expressions are well-formed and have the specified semantics.

```
typename A::element_type
```

2 *Result:* A complete object type that is not an abstract class type.

```
typename A::pointer
```

3 *Result:* A type that models `copyable`, and for which `is_nothrow_move_constructible_v<A::pointer>` is `true`, `is_nothrow_move_assignable_v<A::pointer>` is `true`, and `is_nothrow_swappable_v<A::pointer>` is `true`.

4 [Note: The type of `pointer` need not be `element_type*`. — end note]

```
typename A::reference
```

5 *Result:* A type that models `common_reference_with<A::reference&&, A::element_type&>`.

6 [Note: The type of `reference` need not be `element_type&`. — end note]

```
typename A::offset_policy
```

7 *Result:* A type `OP` such that:

(7.1) — `OP` meets the accessor policy requirements.

(7.2) — `constructible_from<OP, const A&>` is modeled.

(7.3) — `is_same_v<typename OP::element_type, typename A::element_type>` is `true`

```
a.access(p, i)
```

8 *Result:* `A::reference`

9 *Remarks:* The expression is equality preserving.

10 [Note: Concrete accessor policies can impose preconditions for their `access` function. However, they might not. For example, an accessor where `p` is `span<A::element_type,dynamic_extent>` and `access(p,i)` returns `p[i % p.size()]` does not need to impose a precondition on `i`. - end note]

```
a.offset(p, i)
```

11 *Result:* `A::offset_policy::pointer`

12 *Returns:* `q` such that for `b` being `A::offset_policy(a)`, and any integer `n` for which [`0`, `n`) is an accessible range of `p` and `a`:

(12.1) — [`0`, `n-i`) is an accessible range of `q` and `b`, and

(12.2) — `b.access(q, j)` provides access to the same element as `a.access(p, i + j)`, for every `j` in the range [`0`, `n-i`).

13 *Remarks:* The expression is equality preserving.

24.7. .3 Class template `default_accessor` [mdspan.accessor.default]

24.7. .3.1 Overview [mdspan.accessor.default.overview]

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

    constexpr default_accessor() noexcept = default;

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

    constexpr reference access(pointer p, size_t i) const noexcept;

    constexpr pointer offset(pointer p, size_t i) const noexcept;
  };
}
```

1 `default_accessor` meets the accessor policy requirements.

2 `ElementType` is required to be a complete object type that is neither an abstract class type nor an array type.

3 Each specialization of `default_accessor` is a trivially copyable type that models `semiregular`.

4 [0, n) is an accessible range for an object p of type pointer and an object of type default_accessor if and only if [p, p+n) is a valid range.

24.7. .3.2 Members [mdspan.accessor.default.members]

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

1 *Constraints:* is_convertible_v<OtherElementType(*)[], element_type(*)[]> is true.

```
constexpr reference access(pointer p, size_t i) const noexcept;
```

2 *Effects:* equivalent to return p[i];

```
constexpr pointer offset(pointer p, size_t i) const noexcept;
```

3 *Effects:* equivalent to return p + i;.

24.7. Class template mdspan [mdspan.mdspan]

24.7. .1 Overview [mdspan.mdspan.overview]

1 mdspan is a view of a multidimensional array of elements.

```
namespace std {

template<class ElementType, class Extents, class LayoutPolicy, class AccessorPolicy>
class mdspan {
public:
  using extents_type = Extents;
  using layout_type = LayoutPolicy;
  using accessor_type = AccessorPolicy;
  using mapping_type = typename layout_type::template mapping<extents_type>;
  using element_type = ElementType;
  using value_type = remove_cv_t<element_type>;
  using index_type = typename extents_type::index_type ;
  using size_type = typename extents_type::size_type;
  using rank_type = typename extents_type::rank_type ;
  using pointer = typename accessor_type::pointer;
  using reference = typename accessor_type::reference;

  static constexpr rank_type rank() { return extents_type::rank(); }
  static constexpr rank_type rank_dynamic() { return extents_type::rank_dynamic(); }
  static constexpr index_type static_extent(rank_type r) { return extents_type::static_extent(r); }
  constexpr index_type extent(rank_type r) const { return extents().extent(r); }

  // [mdspan.mdspan.ctor], mdspan Constructors
  constexpr mdspan();
  constexpr mdspan(const mdspan& rhs) = default;
  constexpr mdspan(mdspan&& rhs) = default;

  template<class... IndexTypes>
    explicit constexpr mdspan(pointer ptr, IndexTypes... exts);
  template<class IndexType, size_t N>
    explicit(N != rank_dynamic())
    constexpr mdspan(pointer p, span<IndexType, N> exts);
  template<class IndexType, size_t N>
    explicit(N != rank_dynamic())
    constexpr mdspan(pointer p, const array<IndexType, N>& exts);
```

```
constexpr mdspan(pointer p, const extents_type& ext);
constexpr mdspan(pointer p, const mapping_type& m);
constexpr mdspan(pointer p, const mapping_type& m, const accessor_type& a);

template<class OtherElementType, class OtherExtents,
         class OtherLayoutPolicy, class OtherAccessorPolicy>
  explicit(see below)
  constexpr mdspan(
    const mdspan<OtherElementType, OtherExtents,
                 OtherLayoutPolicy, OtherAccessorPolicy>& other);

constexpr mdspan& operator=(const mdspan& rhs) = default;
constexpr mdspan& operator=(mdspan&& rhs) = default;

// [mdspan.mdspan.members], mdspan members
template<class... IndexTypes>
  constexpr reference operator[](IndexTypes... indices) const;
template<class IndexType>
  constexpr reference operator[](span<IndexType, rank()> indices) const;
template<class IndexType>
  constexpr reference operator[](const array<IndexType, rank()>& indices) const;

constexpr index_type size() const;

friend constexpr void swap(
  mdspan<ElementType, Extents, LayoutPolicy, AccessorPolicy>& x,
  mdspan<ElementType, Extents, LayoutPolicy, AccessorPolicy>& y) noexcept;

constexpr const extents_type& extents() const { return map_.extents(); }
constexpr const pointer& data() const { return ptr_; }
constexpr const mapping_type& mapping() const { return map_; }
constexpr const accessor_type& accessor() const { return acc_; }

static constexpr bool is_always_unique() {
  return mapping_type::is_always_unique();
}
static constexpr bool is_always_contiguous() {
  return mapping_type::is_always_contiguous();
}
static constexpr bool is_always_strided() {
  return mapping_type::is_always_strided();
}

constexpr bool is_unique() const {
  return map_.is_unique();
}
constexpr bool is_contiguous() const {
  return map_.is_contiguous();
}
constexpr bool is_strided() const {
  return map_.is_strided();
}
constexpr index_type stride(rank_type r) const {
```

```
    return map_.stride(r);
  }

private:
  accessor_type acc_; // exposition only
  mapping_type map_; // exposition only
  pointer ptr_; // exposition only
};

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

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

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

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

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

}
```

2 *Mandates:*

(2.1) — `ElementType` is a a complete object type that is neither an abstract class type nor an array type,

(2.2) — `Extents` is a specialization of `extents`, and

(2.3) — `is_same_v<ElementType, typename AccessorPolicy::element_type>` is `true`.

3

(3.1) — `LayoutPolicy` shall meet the layout mapping policy requirements [mdspan.layoutpolicy.reqmts], and

(3.2) — `AccessorPolicy` shall meet the accessor policy requirements [mdspan.accessor.reqmts].

4 Each specialization `MDS` of `mdspan` models `copyable` and

(4.1) — `is_nothrow_move_constructible_v<MDS>` is `true`,

(4.2) — `is_nothrow_move_assignable_v<MDS>` is `true`, and

(4.3) — `is_nothrow_swappable_v<MDS>` is `true`.

5 A specialization of `mdspan` is a trivially copyable type if its `accessor_type`, `mapping_type`, and `pointer` are trivially copyable types.

24.7. .2 Constructors [mdspan.mdspan.ctor]

```
constexpr mdspan();
```

1 *Constraints:*

(1.1) — `rank_dynamic() > 0` is `true`.

(1.2) — `is_default_constructible_v<pointer>` is `true`.

(1.3) — `is_default_constructible_v<mapping_type>` is `true`.

(1.4) — `is_default_constructible_v<accessor_type>` is `true`.

2 *Precondition:* [0, *map_*`.required_span_size()`) is an accessible range of *ptr_* and *acc_* for the values of *map_* and *acc_* after the invocation of this constructor.

3 *Effects:* Value-initializes *ptr_*, *map_*, and *acc_*.

```
template<class... IndexTypes>
  explicit constexpr mdspan(pointer p, IndexTypes... exts);
```

4 *Constraints:*

(4.1) — `(is_convertible_v<IndexTypes, index_type> && ...)` is `true`,

(4.2) — `(is_nothrow_constructible<index_type, IndexTypes> && ...)` is `true`,

(4.3) — `(sizeof...(IndexTypes) == rank()) || (sizeof...(IndexTypes) == rank_dynamic())` is `true`,

(4.4) — `is_constructible_v<mapping_type, extents_type>` is `true`, and

(4.5) — `is_default_constructible_v<accessor_type>` is `true`.

5 *Precondition:* [0, *map_*`.required_span_size()`) is an accessible range of `p` and *acc_* for the values of *map_* and *acc_* after the invocation of this constructor.

6 *Effects:*

(6.1) — Direct-non-list-initializes *ptr_* with `std::move(p)`, and

(6.2) — Direct-non-list-initializes *map_* with `extents_type(static_cast<index_type>(std::move(exts))...)`.

(6.3) — Value-initializes *acc_*.

```
template<class IndexType, size_t N>
  explicit(N != rank_dynamic())
  constexpr mdspan(pointer p, span<IndexType, N> exts);
template<class IndexType, size_t N>
  explicit(N != rank_dynamic())
  constexpr mdspan(pointer p, const array<IndexType, N>& exts);
```

7 *Constraints:*

(7.1) — `is_convertible_v<const IndexType&, index_type>` is `true`,

(7.2) — `(is_nothrow_constructible<index_type, const IndexTypes&> && ...)` is `true`,

(7.3) — `(N == rank()) || (N == rank_dynamic())` is `true`,

(7.4) — `is_constructible_v<mapping_type, extents_type>` is `true`, and

(7.5) — `is_default_constructible_v<accessor_type>` is `true`.

8 *Precondition:* [0, *map_*`.required_span_size()`) is an accessible range of `p` and *acc_* for the values of *map_* and *acc_* after the invocation of this constructor.

9 *Effects:*

(9.1) — Direct-non-list-initializes *ptr_* with `std::move(p)`, and

(9.2) — Direct-non-list-initializes *map_* with `extents_type(exts)`.

(9.3) — Value-initializes *acc_*.

```
constexpr mdspan(pointer p, const extents_type& ext);
```

10 *Constraints:*

(10.1) — `is_constructible_v<mapping_type, const extents_type&>` is `true`, and

(10.2) — `is_default_constructible_v<accessor_type>` is `true`.

11 *Precondition:* [0, *map_*`.required_span_size()`) is an accessible range of `p` and *acc_* for the values of *map_* and *acc_* after the invocation of this constructor.

12 *Effects:*

(12.1) — Direct-non-list-initializes *ptr_* with `std::move(p)`, and

(12.2) — Direct-non-list-initializes *map_* with `ext`.

(12.3) — Value-initializes *acc_*.

```
constexpr mdspan(pointer p, const mapping_type& m);
```

13 *Constraints:* `is_default_constructible_v<accessor_type>` is `true`.

14 *Precondition:* [0, `m.required_span_size()`) is an accessible range of `p` and *acc_* for value of *acc_* after the invocation of this constructor.

15 *Effects:*

(15.1) — Direct-non-list-initializes *ptr_* with `std::move(p)`, and

(15.2) — Direct-non-list-initializes *map_* with `m`.

```
constexpr mdspan(pointer p, const mapping_type& m, const accessor_type& a);
```

16 *Precondition:* [0, `m.required_span_size()`) is an accessible range of `p` and `a`.

17 *Effects:*

(17.1) — Direct-non-list-initializes *ptr_* with `std::move(p)`,

(17.2) — Direct-non-list-initializes *map_* with `m`, and

(17.3) — Direct-non-list-initializes *acc_* with `a`.

```
template<class OtherElementType, class OtherExtents,
         class OtherLayoutPolicy, class OtherAccessor>
  explicit(see below)
  constexpr mdspan(const mdspan<OtherElementType, OtherExtents,
                                OtherLayoutPolicy, OtherAccessor>& other);
```

18 *Constraints:*

(18.1) — `is_constructible_v<mapping_type, const OtherLayoutPolicy::template mapping<OtherExtents>&>` is `true`;

(18.2) — `is_constructible_v<accessor_type, const OtherAccessor&>` is `true`; and

19 *Mandates:*

(19.1) — `is_constructible_v<pointer, const OtherAccessor::pointer&>` is `true`;

(19.2) — `is_constructible_v<extents_type, OtherExtents>` is `true`.

20 *Preconditions:*

(20.1) — For each rank index `r` of `extents_type`, `static_extent(r) == dynamic_extent || static_extent(r) == other.exte` is `true`.

(20.2) — [`0`, *`map_`*`.required_span_size()`) is an accessible range of *`ptr_`* and *`acc_`* for values of *`ptr_`*, *`map_`*, and *`acc_`* after the invocation of this constructor.

21 *Effects:*

(21.1) — Direct-non-list-initializes *`ptr_`* with `other.`*`ptr_`*,

(21.2) — Direct-non-list-initializes *`map_`* with `other.`*`map_`*, and

(21.3) — Direct-non-list-initializes *`acc_`* with `other.`*`acc_`*.

22 *Remarks:* The expression inside `explicit` is:

```
!is_convertible_v<const OtherLayoutPolicy::template mapping<OtherExtents>&, mapping_type> ||
!is_convertible_v<const OtherAccessor&, accessor_type>
```

24.7. .3 Members [mdspan.mdspan.members]

```
template<class... IndexTypes>
  constexpr reference operator[](IndexTypes... indices) const;
```

1 *Constraints:*

(1.1) — `(is_convertible_v<IndexTypes, index_type> && ...)` is `true`,

(1.2) — `(is_nothrow_constructible_v<index_type, IndexTypes> && ...)` is `true`, and

(1.3) — `sizeof...(IndexTypes) == rank()` is `true`.

2 Let `I` be `static_cast<index_type>(std::move(indices))`.

3 *Preconditions:* `I` is a multidimensional index in `extents()`. [Note: This implies that *`map_`*`(I...) <`*`map_`*`.required_span_size(` is `true`.— end note];

4 *Effects:* Equivalent to: `return` *`acc_`*`.access(`*`ptr_`*`,`*`map_`*`(I...));`

```
template<class IndexType>
  constexpr reference operator[](span<IndexType, rank()> indices) const;
template<class IndexType>
  constexpr reference operator[](const array<IndexType, rank()>& indices) const;
```

5 *Constraints:*

(5.1) — `is_convertible_v<const IndexType&, index_type>` is `true`, and

(5.2) — `is_nothrow_constructible_v<index_type, const IndexType&>` is `true`.

6 *Effects:* Let `P` be a parameter pack such that `is_same_v<make_index_sequence<rank()>, index_sequence<P...>>` is `true`. Equivalent to: `return operator[](static_cast<index_type>(as_const(indices[P]))...);`

```
constexpr index_type size() const;
```

7 *Precondition:* The size of the multidimensional index space `extents()` is a representable value of type `index_type` ([basic.fundamental]).

8 *Returns:* `extents().`*`fwd-prod-of-extents`*`(rank())`.

```
friend constexpr void swap(
  mdspan<ElementType, Extents, LayoutPolicy, AccessorPolicy>& x,
  mdspan<ElementType, Extents, LayoutPolicy, AccessorPolicy>& y) noexcept;
```