`mdarray`: An Owning Multidimensional Array Analog of `mdspan`

Document #:	P1684R0
Date:	2019-05-28
Project:	Programming Language C++ Library Evolution

Reply-to:	David Hollman <dshollm@sandia.gov> Christian Trott <crtrott@sandia.gov> Mark Hoemmen <mhoemme@sandia.gov> Daniel Sunderland <dsunder@sandia.gov>

Contents

1 Motivation

[P0009R9] introduced a non-owning multidimensional array abstraction that has been refined over many revisions and is expected to be merged into the C++ working draft early in the C++23 cycle. However, there are many owning use cases where `mdspan` is not ideal. In particular, for use cases with small, fixed-size dimensions, the non-owning semantics of `mdspan` may represent a significant pessimization, precluding optimizations that arise from the removal of the non-owning indirection (such as storing the data in registers).

Without `mdarray`, use cases that should be owning are awkward to express:

P0009 Only:

```
void make_random_rotation(mdspan<float, 3, 3> output);
void apply_rotation(mdspan<float, 3, 3>, mdspan<float, 3>);
void random_rotate(mdspan<float, dynamic_extent, 3> points) {
  float buffer[mdspan<float, 3, 3>::required_span_size()] = { };
  auto rotation = mdspan<float, 3, 3>(buffer);
  make_random_rotation(rotation);
  for(int i = 0; i < points.extent(0); ++i) {
    apply_rotation(rotation, subspan(points, i, std::all));
  }
}
```

This Work:

```
mdarray<float, 3, 3> make_random_rotation();
void apply_rotation(mdarray<float, 3, 3>, mdspan<float, 3>);
void random_rotate(mdspan<float, dynamic_extent, 3> points) {
  auto rotation = make_random_rotation();
  for(int i = 0; i < points.extent(0); ++i) {
    apply_rotation(rotation, subspan(points, i, std::all));
  }
}
```

Particularly for small, fixed-dimension `mdspan` use cases, owning semantics can be a lot more convenient and require a lot less in the way of interprocedural analysis to optimize.

2 Design

One major goal of the design for `mdarray` is to parallel the design of `mdspan` as much as possible (but no more), with the goals of reducing cognitive load for users already familiar with `mdspan` and of incorporating the lessons learned from the half decade of experience on [P0009R9]. This paper (and this section in particular) assumes the reader has read and is already familiar with [P0009R9].

2.1 Design Overview

In brief, the analogy to `basic_mdspan` can be seen in the declaration of the proposed design for `basic_mdarray`:

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

This intentionally parallels the design of `basic_mdspan` in [P0009R9], which has the signature:

```
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class AccessorPolicy = accessor_basic<ElementType>>
  class basic_mdspan;
```

The details of this design are included below, along with the accompanying logic and an exploration of alternative designs. In a manner exactly analogous to `mdspan`, we also propose the convenience type alias template `mdarray`, defined as:

```
template <class T, ptrdiff_t... Extents>
  using mdarray = basic_mdarray<T, extents<Extents...>>;
```

2.2 Principal Differences between `mdarray` and `mdspan`

By design, `basic_mdarray` is as similar as possible to `basic_mdspan`, except with container semantics instead of reference semantics. However, the use of container semantics necessitates a few differences. The most notable of these is deep `const`ness. Like all reference semantic types in the standard, `basic_mdspan` has shallow `const`ness, but container types in the standard library propagate `const` through their access functions. Thus, `basic_mdarray` needs `const` and non-`const` versions of every analogous operation in `basic_mdspan` that interacts with the underlying data:

```
template<class ElementType, class Extents, class LayoutPolicy, class ContainerPolicy>
class basic_mdarray {
  /* ... */

  // also in basic_mdspan:
  using pointer = /* ... */;
  // only in basic_mdarray:
  using const_pointer = /* ... */;

  // also in basic_mdspan:
  using reference = /* ... */;
  // only in basic_mdarray:
  using const_reference = /* ... */;

  // analogous to basic_mdspan, except with const_reference return type:
  constexpr const_reference operator[](index_type) const;
  template<class... IndexType>
    constexpr const_reference operator()(IndexType...) const;
  template<class IndexType, size_t N>
    constexpr const_reference operator()(const array<IndexType, N>&) const;
  // non-const overloads only in basic_mdarray:
  constexpr reference operator[](index_type);
  template<class... IndexType>
    constexpr reference operator()(IndexType...);
  template<class IndexType, size_t N>
    constexpr reference operator()(const array<IndexType, N>&);

  // also in basic_mdspan, except with const_pointer return type:
  constexpr const_pointer data() const noexcept;
  // non-const overload only in basic_mdarray:
  constexpr pointer data() noexcept;

  /* ... */
};
```

Additionally, `basic_mdarray` needs a means of interoperating with `basic_mdspan` in roughly the same way

as contiguous containers interact with `span`, or as `string` interacts with `string_view`. We could do this by adding a constructor to `basic_mdspan`, which would be more consistent with the analogous features in `span` and `string_view`, but in the interest of avoiding modifications to an in-flight proposal, we propose using a member function of `basic_mdarray` for this functionality for now (tentatively named `view()`, subject to bikeshedding). We are happy to change this based on design direction from LEWG:

```
template<class ElementType, class Extents, class LayoutPolicy, class ContainerPolicy>
class basic_mdarray {
  /* ... */

  // only in basic_mdarray:
  using view_type = /* ... */;
  using const_view_type = /* ... */;
  view_type view() noexcept;
  const_view_type view() const noexcept;

  /* ... */
};
```

As discussed below, `accessor_policy` from `basic_mdspan` is replaced by `container_policy` in `basic_mdarray`:

```
template<class ElementType, class Extents, class LayoutPolicy, class ContainerPolicy>
class basic_mdarray {
  /* ... */

  // only in basic_mdarray:
  using container_policy_type = ContainerPolicy;
  using container_type = /* ... */;

  /* ... */
};
```

2.2.0.1 Constructors and Assignment Operators

There are several relatively trivial differences between `basic_mdspan` and `basic_mdarray` constructors and assignment operators. Most trivially, `basic_mdspan` provides a compatible `basic_mdspan` copy-like constructor and copy-like assignment operator, with proper constraints and expectations to enforce compatibility of shape, layout, and size. Since `basic_mdarray` has owning semantics, we also need move-like versions of these:[1]

```
template<class ElementType, class Extents, class LayoutPolicy, class ContainerPolicy>
class basic_mdarray {
  /* ... */

  // analogous to basic_mdspan:
  template<class ET, class Exts, class LP, class CP>
    constexpr basic_mdarray(const basic_mdarray<ET, Exts, LP, CP>&);
  template<class ET, class Exts, class LP, class CP>
    constexpr basic_mdarray& operator=(const basic_mdarray<ET, Exts, LP, CP>&);
  // only in basic_mdarray:
  template<class ET, class Exts, class LP, class CP>
```

[1] Arguably, `basic_mdspan` should also have these constructors because of the potential cost of copying stateful mapping and accessor policy members, but we are not proposing that change here.

```
    constexpr basic_mdarray(basic_mdarray<ET, Exts, LP, CP>&&) noexcept(see-below);
  template<class ET, class Exts, class LP, class CP>
    constexpr basic_mdarray& operator=(basic_mdarray<ET, Exts, LP, CP>&&) noexcept;

  /* ... */
};
```

(The `noexcept` clauses on these constructors and operators should probably actually derive from `noexcept` clauses on the analogous functionality for the element type and policy types).

Additionally, the analog of the `basic_mdspan(pointer, IndexType...)` constructor for `basic_mdarray` should not take the first argument, since the `basic_mdarray` owns the data and thus should be able to construct it from sizes:

```
template<class ElementType, class Extents, class LayoutPolicy, class ContainerPolicy>
class basic_mdarray {
  /* ... */

  // only in basic_mdarray
  template <class... IndexType>
  explicit constexpr basic_mdarray(IndexType...);

  /* ... */
};
```

Note that in the completely static extents case, this is ambiguous with the default constructor. For consistency in generic code, the semantics of this constructor should be preferred over those of the default constructor in that case.

There is some question as to whether we should also have constructors that take `container_type` instances in addition to indices. Consistency with standard container adapters like `std::priority_queue` would dictate that we should; however allowing this would prevent the `ContainerPolicy` (discussed below) from having full control over the container creation process. For simplicity, we omit these constructors for now, leaving the question open to further discussion.

By this same logic, we arrive at the `mapping_type` and `container_policy` constructor analogs:

```
template<class ElementType, class Extents, class LayoutPolicy, class ContainerPolicy>
class basic_mdarray {
  /* ... */

  // only in basic_mdarray
  explicit constexpr basic_mdarray(const mapping_type&);
  constexpr basic_mdarray(const mapping_type&, const container_policy_type&);

  /* ... */
};
```

Finally, we remmove the constructor that takes an `array<IndexType, N>` of dynamic extents because of the possible ambiguity or confusion with a (potential future work) constructor that takes a container instance in the case where the `container_type` happens to be an `array<IndexType, N>`.

2.3 `Extents` Design Reused

As with `basic_mdspan`, the `Extents` template parameter to `basic_mdarray` shall be a template instantiation of `std::extents`, as described in [P0009R9]. The concerns addressed by this aspect of the design are exactly the same in `basic_mdarray` and `basic_mdspan`, so using the same form and mechanism seems like the right thing to do here.

2.4 `LayoutPolicy` Design Reused

While not quite as straightforward, the decision to use the same design for `LayoutPolicy` from `basic_mdspan` in `basic_mdarray` is still quite obviously the best choice. The only piece that's a bit of a less perfect fit is the `is_contiguous()` and `is_always_contiguous()` requirements. While non-contiguous use cases for `basic_mdspan` are quite common (e.g., `subspan()`), non-contiguous use cases for `basic_mdarray` are expected to be a bit more arcane. Nonetheless, reasonable use cases do exist (for instance, padding of the fast-running dimension in anticipation of a resize operation), and the reduction in cognitive load due to concept reuse certainly justifies reusing `LayoutPolicy` for `basic_mdarray`.

2.5 `AccessorPolicy` Replaced by `ContainerPolicy`

By far the most complicated aspect of the design for `basic_mdarray` is the analog of the `AccessorPolicy` in `basic_mdspan`. The `AccessorPolicy` for `basic_mdspan` is clearly designed with non-owning semantics in mind–it provides a `pointer` type, a `reference` type, and a means of converting from a pointer and an offset to a reference. Beyond the lack of an allocation mechanism (that would be needed by `basic_mdarray`), the `AccessorPolicy` requirements address concerns normally addressed by the allocation mechanism itself. For instance, the C++ named requirements for `Allocator` allow for the provision of the `pointer` type to `std::vector` and other containers. Arguably, consistency between `basic_mdarray` and standard library containers is far more important than with `basic_mdspan` in this respect. Several approaches to addressing this incongruity are discussed below.

2.5.1 Expected Behavior of Motivating Use Cases

Regardless of the form of the solution, there are several use cases where we have a clear understanding of how we want them to work. As alluded to above, perhaps *the* most important motivating use case for `mdarray` is that of small, fixed-size extents. Consider a fictitious (not proposed) function, *get-underlying-container*, that somehow retrieves the underlying storage of an `mdarray`. For an `mdarray` of entirely fixed sizes, we would expect the default implementation to return something that is (at the very least) convertible to `array` of the correct size:

```
auto a = mdarray<int, 3, 3>();
std::array<int, 9> data = get-underlying-container(a);
```

(Whether or not a reference to the underlying container should be obtainable is slightly less clear, though we see no reason why this should not be allowed.) The default for an `mdarray` with variable extents is only slightly less clear, though it should almost certainly meet the requirements of *contiguous container* ([**container.requirements.general**]/13). The default model for *contiguous container* of variable size in the standard library is `vector`, so an entirely reasonable outcome would be to have:

```
auto a = mdarray<int, 3, dynamic_extent>();
std::vector<int> data = get-underlying-container(a);
```

Moreover, taking a view of a `basic_mdarray` should yield an analogous `basic_mdspan` with consistent semantics (except, of course, that the latter is non-owning). We provisionally call the method for obtaining this analog `view()`:

```
template <class T, class Extents, class LayoutPolicy, class ContainerPolicy>
void frobnicate(basic_mdarray<T, Extents, LayoutPolicy, ContainerPolicy> data)
{
  auto data_view = data.view();
  /* ... */
}
```

Using `data_view` should be analogous to using `data` in most ways (with the exception of things that relate to ownership, like the copy constructor). This means that `decltype(data_view)::accessor_policy` should implement the same access semantics as `ContainerPolicy`. For `ContainerPolicy`s provided by the standard, this is not a problem, but for an arbitrary `ContainerPolicy`, we probably require more information than is provided by the `Container` concept or any of its refinements. (See below for more discussion of "probably").

The tension between consistency with existing standard library container concepts and interoperability with `basic_mdspan` is the crux of the design challenge for the `ContainerPolicy` customization point (and, indeed, for `mdarray` as a whole).

2.5.2 Analogs in the Standard Library: Container Adapters

Perhaps the best analogs for what `basic_mdarray` is doing with respect to allocation and ownership are the container adaptors ([**container.adaptors**]), since they imbue additional semantics to what is otherwise an ordinary container. These all take a `Container` template parameter, which defaults to `deque` for `stack` and `queue`, and to `vector` for `priority_queue`. The allocation concern is thus delegated to the container concept, reducing the cognitive load associated with the design. While this design approach overconstrains the template parameter slightly (i.e., not all of the requirements of the `Container` concept are needed by the container adaptors), the simplicity arising from concept reuse more than justifies the cost of the extra constraints.

It is difficult to say whether the use of `Container` directly, as with the container adaptors, is also the correct approach for `basic_mdarray`. There are pieces of information that may need to be customized in some very reasonable use cases that are not provided by the standard container concept. The most important of these is the ability to produce a semantically consistent `AccessorPolicy` when creating a `basic_mdspan` that refers to a `basic_mdarray`. (Interoperability between `basic_mdspan` and `basic_mdarray` is considered a critical design requirement because of the nearly complete overlap in the set of algorithms that operate on them.) For instance, given a `Container` instance `c` and an `AccessorPolicy` instance `a`, the behavior of `a.access(p, n)` should be consistent with the behavior of `c[n]` for a `basic_mdspan` wrapping `a` that is a view of a `basic_mdarray` wrapping `c` (if `p` is `c.begin()`). But because `c[n]` is part of the container requirements and thus may encapsulate any arbitrary mapping from an offset of `c.begin()` to a reference, the only reasonable means of preserving these semantics is to reference the original container directly in the corresponding `AccessorPolicy`. In other words, the signature for the `view()` method of `mdarray` would need to look something like (ignoring, for the moment, whether the name for the type of the accessor is specified or implementation-defined):

```
template<class ElementType,
         class Extents,
         class LayoutPolicy,
         class Container>
struct basic_mdarray {
 /* ... */
```

```
 basic_mdspan<
   ElementType, Extents, LayoutPolicy,
   container_reference_accessor<Container>>
 view() const noexcept;
 /* ... */
};

template <class Container>
struct __container_reference_accessor { // not proposed
  using pointer = Container*;
  /* ... */
  template <class Integer>
  reference access(pointer p, Integer offset) {
    return (*p)[offset];
  }
  /* ... */
};
```

But this approach comes at the cost of an additional indirection (one for the pointer to the container, and one for the container dereference itself), which is likely unacceptable cost in a facility designed to target performance-sensitive use cases. The situation for the `offset` requirement (which is used by `subspan`) is potentially even worse for arbitrary non-contiguous containers, adding up to one indirection per invocation of `subspan`. This is likely unacceptable in many contexts.

Nonetheless, using refinements of the existing `Container` concept directly with `basic_mdarray` is an incredibly attractive option because it avoids the introduction of an extra concept, and thus significantly decreases the cognitive cost of the abstraction. Thus, direct use of the existing `Container` concept hierarchy should be preferred to other options unless the shortcomings of the existing concept are so irreconcilable (or so complicated to reconcile) as to create more cognitive load than is needed for an entirely new concept.

2.5.3 (Not Proposed) Alternative: A Dedicated `ContainerPolicy` Concept

Despite the additional cognitive load, there are a few arguments in favor of using a dedicated concept for the container description of `basic_mdarray`. As is often the case with concept-driven design, the implementation of `basic_mdarray` only needs a relatively small subset of the interface elements in the `Container` concept hierarchy. This alone is not enough to justify an additional concept external to the existing hierarchy; however, there are also quite a few features missing from the existing container concept hierarchy, without which an efficient `basic_mdarray` implementation may be difficult or impossible. As alluded to above, conversion to an `AccessorPolicy` for the creation of a `basic_mdspan` is one missing piece. (Another, interestingly, is sized construction of the container mixed with allocator awareness, which is surprisingly lacking in the current hierarchy somehow.) For these reasons, it is worth exploring a design based on analogy to the `AccessorPolicy` concept rather than on analogy to `Container`. If we make that abstraction owning, we might call it something like `_ContainerLikeThing` (not proposed here; included for discussion). In that case, a model of the `_ContainerLikeThing` concept that meets the needs of `basic_mdarray` might look something like:

```
template <class ElementType, class Allocator=std::allocator<ElementType>>
struct vector_container_like_thing // models _ContainerLikeThing
{
public:
  using element_type = ElementType;
  using container_type = std::vector<ElementType, Allocator>;
```

```
  using allocator_type = typename container_type::allocator_type;
  using pointer = typename container_type::pointer;
  using const_pointer = typename container_type::const_pointer;
  using reference = typename container_type::reference;
  using const_reference = typename container_type::const_reference;
  using accessor_policy = std::accessor_basic<element_type>;
  using const_accessor_policy = std::accessor_basic<const element_type>;

  // analogous to `access` method in `AccessorPolicy`
  reference access(ptrdiff_t offset) { return __c[size_t(offset)]; }
  const_reference access(ptrdiff_t offset) const { return __c[size_t(offset)]; }

  // Interface for mdspan creation
  accessor_policy make_accessor_policy() { return { }; }
  const_accessor_policy make_accessor_policy() const { return { }; }
  typename pointer data() { return  __c.data(); }
  typename const_pointer data() const { return  __c.data(); }

  // Interface for sized construction
  static vector_container_policy create(size_t n) {
    return vector_container_like_thing{container_type(n, element_type{})};
  }
  static vector_container_policy create(size_t n, allocator_type const& alloc) {
    return vector_container_like_thing{container_type(n, element_type{}, alloc)};
  }

  container_type __c;
};
```

This approach solves many of the problems associated with using the `Container` concept directly. It is the most flexible and provides the best compatibility with `mdspan`, since the conversion to analogous `AccessorPolicy` is fully customizable. This comes at the cost of additional cognitive load, but this can be justified based on the observation that almost half of the functionality in the above sketch is absent from the container hierarchy: the `make_accessor_policy()` requirement and the sized, allocator-aware container creation (`create(n, alloc)`) have no analogs in the container concept hierarchy. Non-allocator-aware creation (`create(n)`) is analogous to sized construction from the sequence container concept, the `data()` method is analogous to `begin()` on the contiguous container concept, and `access(n)` is analogous to `operator[]` or `at(n)` from the optional sequence container requirements. Even for these latter pieces of functionality, though, we are required to combine several different concepts from the `Container` hierarchy. Based on this analysis, we have decided it is reasonable to pursue designs for this customization point that diverge from `Container`, including ones that use `AccessorPolicy` as a starting point. Given a better design, we would definitely consider reversing direction on this decision, but despite significant effort, we were unable to find a design that was more than an awkward and forced fit for the `Container` concept hierarchy.

2.5.4 Proposed Alternative: `ContainerPolicy` subsumes `AccessorPolicy`

The above approach has the significant drawback that the `_ContainerLikeThing` is an owning abstraction fairly similar to a container that diverges from the `Container` hierarchy. We initially explored this direction because it avoids having to provide a `basic_mdarray` constructor that takes both a `Container` and a `ContainerPolicy`, which we felt was a "design smell." Another alternative along these lines is to make the `basic_mdarray` itself own the container instance and have the `ContainerPolicy` (name sub-

ject to bikeshedding; maybe `ContainerFactory` or `ContainerAccessor` is more appropriate?) be a non-owning abstraction that describes the container creation and access. While this approach leads to an ugly `basic_mdarray(container_type, mapping_type, ContainerPolicy)` constructor, the analog that constructor affords to the `basic_mdspan(pointer, mapping_type, AccessorPolicy)` constructor is a reasonable argument in favor of this design despite its quirkiness. Furthermore, this approach affords the opportunity to explore a `ContainerPolicy` design that subsumes `AccessorPolicy`, thus providing the needed conversion to `AccessorPolicy` for the analogous `basic_mdspan` by simple subsumption. More importantly, this subsumption would significantly decrease the cognitive load for users already familiar with `basic_mdspan`. A model of `ContainerPolicy` for this sort of approach might look something like:

```
template <class ElementType, class Allocator=std::allocator<ElementType>>
struct vector_container_policy // models ContainerPolicy (and thus AccessorPolicy)
{
public:
  using element_type = ElementType;
  using container_type = std::vector<ElementType, Allocator>;
  using allocator_type = typename container_type::allocator_type;
  using pointer = typename container_type::pointer;
  using const_pointer = typename container_type::const_pointer;
  using reference = typename container_type::reference;
  using const_reference = typename container_type::const_reference;
  using offset_policy = vector_container_policy<ElementType, Allocator>

  // ContainerPolicy requirements:
  reference access(container_type& c, ptrdiff_t i) { return c[size_t(i)]; }
  const_reference access(container_type const& ptrdiff_t i) const { return c[size_t(i)]; }

  // ContainerPolicy requirements (interface for sized construction):
  container_type create(size_t n) {
    return container_type(n, element_type{});
  }
  container_type create(size_t n, allocator_type const& alloc) {
    return container_type(n, element_type{}, alloc);
  }

  // AccessorPolicy requirement:
  reference access(pointer p, ptrdiff_t i) { return p[i]; }
  // For the const analog of AccessorPolicy:
  const_reference access(const_pointer p, ptrdiff_t i) const { return p[i]; }

  // AccessorPolicy requirement:
  pointer offset(pointer p, ptrdiff_t i) { return p + i; }
  // For the const analog of AccessorPolicy:
  const_pointer offset(const_pointer p, ptrdiff_t i) const { return p + i; }

  // AccessorPolicy requirement:
  element_type* decay(pointer p) { return p; }
  // For the const analog of AccessorPolicy:
  const element_type* decay(pointer p) const { return p; }

};
```

The above sketch makes clear the biggest challenge with this approach: the mismatch in shallow versus deep `const`ness in for an abstractions designed to support `basic_mdspan` and `basic_mdarray`, respectively. The `ContainerPolicy` concept thus requires additional `const`-qualified overloads of the basis operations. Moreover, while the `ContainerPolicy` itself can be obtained directly from the corresponding `AccessorPolicy` in the case of the non-`const` method for creating the corresponding `basic_mdspan` (provisionally called `view()`), the `const`-qualified version needs to adapt the policy, since the nested types have the wrong names (e.g., `const_pointer` should be named `pointer` from the perspective of the `basic_mdspan` that the `const`-qualified `view()` needs to return). This could be fixed without too much mess using an adapter (that does not need to be part of the specification):

```
template <ContainerPolicy P>
class __const_accessor_policy_adapter { // models AccessorPolicy
public:
  using element_type = add_const_t<typename P::element_type>;
  using pointer = typename P::const_pointer;
  using reference = typename P::const_reference;
  using offset_policy = __const_accessor_policy_adapter<typename P::offset_policy>;

  reference access(pointer p, ptrdiff_t i) { return acc_.access(p, i); }
  pointer offset(pointer p, ptrdiff_t i) { return acc_.offset(p, i); }
  element_type* decay(pointer p) { return acc_.decay(p); }

private:
  [[no_unique_address]] add_const_t<P> acc_;
};
```

We feel that this approach provides the best balance of cognitive load and flexibility for designs that meet the requirements of this customization point for `basic_mdarray`. Because of this and the other arguments discussed above, we have decided to proceed with this design for `ContainerPolicy` at this point.

3 Synopsis

In the interest of promoting further discussion in design review, a synopsis of the final `basic_mdarray` class template is included here. Much of the design is identical or analogous to `basic_mdspan`, with most of the important differences already discussed above.

```
template<class ElementType, class Extents, class LayoutPolicy, class ContainerPolicy>
class basic_mdarray {
public:

  // Domain and codomain types (also in basic_mdspan)
  using extents_type = Extents;
  using layout_type = LayoutPolicy;
  using mapping_type = typename layout_type::template mapping_type<extents_type>;
  using element_type = typename container_policy::element_type;
  using value_type = remove_cv_t<element_type>;
  using index_type = ptrdiff_t;
  using difference_type = ptrdiff_t;
  using pointer = typename container_policy::pointer;
  using reference = typename container_policy::reference;
```

```
// Domain and codomain types (unique to basic_mdarray)
using container_policy_type = ContainerPolicy;
using container_type = typename container_policy_type::container_type;
using const_pointer = typename container_policy_type::const_pointer;
using const_reference = typename container_policy_type::const_reference;
using view_type =
  basic_mdspan<element_type, extents_type, layout_type, ContainerPolicy>;
using const_view_type =
  basic_mdspan<const element_type, extents_type, layout_type, see-below>;

// basic_array constructors, assignment, and destructor
constexpr basic_mdarray() noexcept = default;
constexpr basic_mdarray(const basic_mdarray&) noexcept = default;
constexpr basic_mdarray(basic_mdarray&&) noexcept = default;
template<class... IndexType>
  explicit constexpr basic_mdarray(IndexType... dynamic_extents);
explicit constexpr basic_mdarray(const mapping_type& m);
constexpr basic_mdarray(const mapping_type& m, const container_policy_type& p);
template<class ET, class Exts, class LP, class CP>
  constexpr basic_mdarray(const basic_mdarray<ET, Exts, LP, CP>& other);
template<class ET, class Exts, class LP, class CP>
  constexpr basic_mdarray(basic_mdarray<ET, Exts, LP, CP>&& other);

~basic_mdarray() = default;

constexpr basic_mdarray& operator=(const basic_mdarray&) noexcept = default;
constexpr basic_mdarray& operator=(basic_mdarray&&) noexcept = default;
template<class ET, class Exts, class LP, class CP>
  constexpr basic_mdarray& operator=(const basic_mdarray<ET, Exts, LP, CP>&) noexcept;
template<class ET, class Exts, class LP, class CP>
  constexpr basic_mdarray& operator=(basic_mdarray<ET, Exts, LP, CP>&&) noexcept;

// basic_mdarray mapping domain multidimensional index to access codomain
// element (also in basic_mdspan)
constexpr reference operator[](index_type);
constexpr const_reference operator[](index_type) const;
template<class... IndexType>
  constexpr reference operator()(IndexType... indices);
template<class... IndexType>
  constexpr const_reference operator()(IndexType... indices) const;
template<class IndexType, size_t N>
  constexpr reference operator()(const array<IndexType, N>& indices);
template<class IndexType, size_t N>
  constexpr const_reference operator()(const array<IndexType, N>& indices) const;

// basic_mdarray observers of the domain multidimensional index space
// (also in basic_mdspan)
static constexpr int rank() noexcept;
static constexpr int rank_dynamic() noexcept;
static constexpr index_type static_extent(size_t) noexcept;
constexpr extents_type extents() const noexcept;
```

```
  constexpr index_type extent(size_t) const noexcept;
  constexpr index_type size() const noexcept;
  constexpr index_type unique_size() const noexcept;

  // creation of analogous mdspan (unique to basic_mdarray)
  constexpr view_type view() noexcept;
  constexpr const_view_type view() const noexcept;

  // observers of the codomain (also in basic_mdspan)
  constexpr pointer data() noexcept;
  constexpr const_pointer data() const noexcept;
  constexpr container_policy_type container_policy() const noexcept;

  // basic_mdarray observers of the mapping (also in basic_mdspan)
  static constexpr bool is_always_unique() noexcept;
  static constexpr bool is_always_contiguous() noexcept;
  static constexpr bool is_always_strided() noexcept;
  constexpr mapping_type mapping() const noexcept;
  constexpr bool is_unique() const noexcept;
  constexpr bool is_contiguous() const noexcept;
  constexpr bool is_strided() const noexcept;
  constexpr index_type stride(size_t) const;

private:
  container_type c_; // exposition only
  mapping_type map_; // exposition only
  container_policy cp_; // exposition only
};
```

4 Wording

Wording will be added after design review is completed and after wording review is completed for [P0009R9], but most of the challenges associated with wording for `basic_mdarray` have already been solved in the context of [P0009R9].

5 References

[P0009R9] H. Carter Edwards, Bryce Adelstein Lelbach, Daniel Sunderland, David Hollman, Christian Trott, Mauro Bianco, Ben Sander, Athanasios Iliopoulos, John Michopoulos, Mark Hoemmen. 2019. mdspan: A Non-Owning Multidimensional Array Reference.
https://wg21.link/p0009r9