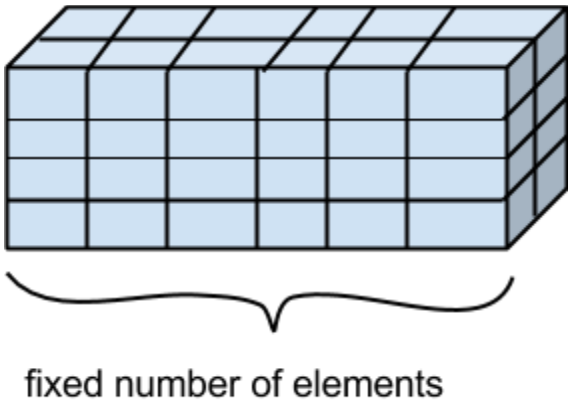
This piece is ~~stolen~~ adapted from Section 7.2 of *The C++ Standard Library: A Tutorial and Reference, second edition*, by Nicolai M. Josuttis.

Multi-Dimensional Arrays

An instance of the container class `array_md<>` models a static (nested) array. It wraps an ordinary static C-style array providing the interface of an STL container. Conceptually, an array is a sequence of elements with constant size. Thus, you can neither add nor remove elements to change the size. Only a replacement of element values is possible.



Class `array_md<>`, the main type of this library submission, results from extending the `std::array<>` class with the new (to C++2011) feature of variadic templates. The idea also appeared in the first proposal on variadic templates [WG21: N2080=06-0150, section 2.1]. It is safer and has no worse performance than an ordinary (nested) array.

To use a multi-dimensional array, you must include the header file:

```
#include <boost/container/array_md.hpp>
```

There, the type is defined as a class template:

```
namespace boost {
namespace container {
     template <typename Element, std::size_t ...Extents>
     struct array_md;
} }
```

The elements of an array may be of any type allowed in C-level arrays.

If there are any template parameters after the first, they specify the index range for a given dimension. For example, the type `array_md<T, 6, 7>` can have its first index go from 0 to 5, and its second index go from 0 to 6. These extents determine the number of elements the array has throughout its lifetime. Thus, `size()` always yields the product of the extents. (It's the multiplicative identity, i.e. 1, when no extents are given.)

Allocator support is not provided.

Abilities of Arrays

Arrays copy their elements into their internal static C-style array. The elements always have fixed positions, and those have a standardized order. Thus, arrays are a kind of *ordered collection*. Arrays provide *random access*. Thus, you can access every element directly in constant time, provided that you know its position. The iterators are random-access iterators, so you can use any algorithm of the STL.

If you need a sequence with a fixed number of elements, class `array_md<>` has the best performance because memory is allocated on the stack (if possible), reallocation never happens, and you have random access.

Initialization

Regarding initialization, class `array_md<>` has some unique semantics. As a first example, the default constructor does not create an empty container, because the number of elements in the container is always constant according to the non-first template parameters throughout its lifetime.

Note that `array_md<>`, like `std::array<>`, has its elements default initialized when nothing is passed to initialize the elements. This means that for fundamental types and other trivial types, the initial value might be undefined rather than zero. For example:

```
array_md<int, 2, 2>  x;      // OOPS: elements of x have undefined value
```

You can provide an empty initializer list instead. In that case, all values are guaranteed to be value initialized, which has the effect that elements of fundamental types are zero initialized:

```
array_md<int, 2, 2>  x = {};      // OK: all elements of x have value 0 (int{})
```

The reason is that `array_md<>` counts as an aggregate type, and therefore can use aggregate

initialization[1]. The format of the initializer can match with what works with a structure enclosing a nested array (or a single object if zero dimensions are used).

If an initializer format misses some elements, those elements will be value initialized.

```
array_md<int, 10>    c1 = { 42 };      // First element has value 42,
                                       // followed by 9 elements with value 0.
array_md<int, 3, 3>  c2 = {{1,2,3}}; // First sub-array has values 1, 2, and 3;
                                       // followed by 2 zeroed sub-arrays.
array_md<int, 6, 6>  c3 = {{4, 5}};   // First two elements have values 4 and 5,
                                       // while the remainder are zeroed.
```

If there are too many elements in the initializer, in whole or part, that's ill-formed:

```
array_md<int, 2>     c4 = { {1, 2, 3} };    // ERROR: too many values
array_md<int, 2, 2>  c5 = { {{4, 5}, {6, 7, 8}, {}} };
      // Excessive initializers in two places (inner array and second sub-array).
```

A common format that works for all dimensions and sizes is a single depth list, using braces elision:

```
array_md<int, 5>          d1{ 1, 2, 3, 4, 5 };
array_md<int, 2, 2>       d2{ 1, 2, 3 };
array_md<char[6], 2, 2>   d3{ "Hello", "World", "Debit", "Array" };
array_md<char[3], 3>      d4{ 'I', 't', '\0', 'i', 's', '\0', 'o',
 'n', '\0' };
```

Since `array_md<>` uses a C-level (nested) array, the initializer can explode element types that are themselves arrays. The element correspondences is the usual "row-major," where the rightmost extent varies the fastest, order that the layout rules impose.

Other effects of making `array_md<>` an aggregate is that regular parenthesis-based constructor syntax containing the elements isn't well-formed. Also, the internal array holding the elements is a public non-static class member, but its name can be avoided by other, more portable ways to address that subobject in part or total.

`swap()` and Move Semantics

As for all other containers, `array_md<>` provides `swap()` operations. Thus, you can swap elements with a container of the same type (same element type, number of dimensions, and

[1] The initializer list is a generalization of this concept to scalars and non-aggregate class-types.

same corresponding extents). Note, however, that an `array_md<>` can't simply swap pointers internally. For this reason, `swap()` has linear complexity[2] and the effect that iterators and references don't swap containers with their elements. So, iterators and references refer to the same container but different elements afterward.

You can use move semantics, which are implicitly provided for arrays as long as the element type supports them. For example:

```
array_md<std::string, 10, 20>  as1, as2
//...
as1 = std::move( as2 );
```

Size

Unlike `std::array<>`, it is not possible to 0 as a value of any of the array's extents. Although that is inconsistent with `std::array<>`, it is consistent with C-level arrays[3]. It is more complicated also since there would be multiple points to introduce a zero-valued parameter, and mess up the recursive building technique to implement `array_md<>`. It keeps the total number of elements non-decreasing as dimensions are added.

An advantage is that the array is never empty, and so `begin()`, `cbegin()`, `rbegin()`, and `crbegin()` are always valid. And their return values are always distinct from their corresponding past-the-end member functions' returns. And that `front()` and `back()` can always be called, and that `data()` always returns a valid pointer.

An array's size is fixed at compile-time, so the total number of elements is exposed through a constant expression in a static data member called `static_size`. The number of dimensions is in a `static constexpr` member called `dimensionality`. If that member is non-zero, then there is a `static constexpr` array `static_sizes` listing each dimension's extent, in template declaration order.

Types

The type of the elements is exposed through the type-alias `value_type`. The other container and allocator types are provided: `size_type`, `difference_type`, `reference`, `const_reference`, `iterator`, `const_iterator`, `pointer`, and `const_pointer`.

---

[2] It's linear if you compare to containers that can vary their length. Since an `array_md<>` has a fixed number of elements, the number of swaps is always constant!

[3] And like C-level arrays, the element type can't be a reference, function reference, nor a class-type that's considered abstract.

The type of the internal data object is aliased as `data_type`. It is the same as `value_type` when `dimensionality` is zero, and a (nested) array of `value_type` otherwise. For the non-zero dimension cases, the immediate element type of the top-level internal data array is aliased as `direct_element_type`. That type is the same as `value_type` only when `dimensionality` is one, and some intermediate (nested) array of `value_type` when it's at least two.

Array Operations

Create, Copy, and Destroy

Since `array_md<>` is an aggregate class, its special member functions (default construction, copy construction, move construction, destruction, copy-assignment, and move-assignment) are defined if the corresponding operation for `value_type` (after stripping extents if it's itself an array type) is defined. The default constructor uses default initialization on the elements, so they will have undefined values if the elements are of fundamental or otherwise trivial type. If an initializer list is used without enough elements, including being empty, then the remaining elements are value initialized (0 for fundamental types).

Again, note that unlike with other containers, you can't use the parenthesis syntax with initializer lists:

```
array_md<int, 5>  a( {1, 2, 3, 4, 5} );    // ERROR
```

Non-Modifying Operations

Operation	Effect
`a.empty()`	Returns whether the array is empty. May be faster than `a.size() == 0`. Always `false` for arrays.
`a.size()`	Returns the current number of elements. Always fixed as `static_size`.
`a.max_size()`	Returns the maximum number of elements possible. Always fixed as `static_size`.
`a1 == a2`	Returns whether a1 is equal to a2. It calls == for the element types.

`a1 != a2`	Returns whether a1 is not equal to a2. Computed as `!(a1 == a2)`.
`a1 < a2`	Returns whether a1 is less than a2. Calls < and == for the element types.
`a1 > a2`	Returns whether a1 is greater than a2. Computed as `a2 < a1`.
`a1 <= a2`	Returns whether a1 is less than or equal to a2. Computed as `!(a2 < a1)`.
`a1 >= a2`	Returns whether a1 is greater than or equal to a2. Computed as `!(a1 < a2)`.

Both equality and ordered comparisons require the operands to have the same dimension count and corresponding extents. The element types of the operands, which may differ, have to have a well-defined operator == and/or < between them[4]. Ordered comparisons between operands are lexicographic, even when the dimensions count exceeds one. The lexicographic traversal is defined as starting from the element whose index position is all zeroes, proceeding through the row-based order with the rightmost indices varying the fastest. (When the array is dimensionless, then the traversal touches just the sole element.)

Assignments

Besides the automatically-defined copy- and move-assignment, there is a member-function and free-function `swap()` that take two arrays as operand/arguments. The two arrays in these cases have to have the same type (element type, dimension count, and corresponding extents).

The `swap()` routines cannot guarantee small-constant complexity for arrays, since elements have to be swapped wholesale instead of a few owning pointers. Elements keep their addresses and owning container and change values (instead of the usual container swapping that make elements switch their owning container and keep their addresses and values).

There is also the `fill()` member function that takes one argument of `value_type`. That argument is assigned to all the elements of the array. This normally requires `value_type` to be Assignable, which won't work if that type is itself an array. So `fill()` assigns with a procedure that copies memberwise (recursively) if needed. So the requirement is that `value_type` *after* all array-extents are removed has to be Assignable.

Internally, all these operations call the assignment operator of the (extents-stripped) element type.

[4] When both operands' element types are C-level arrays, the comparisons will occur after array-to-pointer decay, making the results meaningless (even assuming they compile).

Element Access

To access all elements of an array, you must use range-based `for` loops, specific operations, or iterators. In addition, a tuple interface is provided, so you can also use `get<>()` to access a specific element. The `get` functions use a single number as an index, that from the lexicographic traversal order (so the gotten type is always `value_type`). The following table list the standard ways to directly access an element, for an `array_md<>` with *N* + 1 dimensions:

Operation	Effect
`a[ i0 ]`	Returns the top-level sub-object with index *i0* (**no** range checking). The sub-object is an element when `dimensionality` is 1, and a sub-array when greater than 1. This member function is not present when `dimensionality` is 0.
`a( i0, …, iN )`	Returns the element with *i0* for the first index, through *iN* for the last index (**no** range checking)
`a.at( i0, …, iN )`	Returns the element with *i0* for the first index, through *iN* for the last index (throwing a range-error exception if at least one index is out-of-range).
`a[ {i0, …, iN} ]`	Returns the element with *i0* for the first index, through *iN* for the last index (**no** range checking), all passed through an initializer list. This member function is not present when `dimensionality` is 0.
`a( {i0, …, iN} )`	Same as above, except it is present even when `dimensionality` is 0.
`a.at( {i0, …, iN} )`	Returns the element with *i0* for the first index, through *iN* for the last index (throwing exceptions if any index is out-of-range, or if the number of indices differs from `dimensionality`), all passed through an initializer list.
`a.front()`	Returns the first element
`a.back()`	Returns the last element

When the array's `dimensionality` is greater than 1, the sub-array returned by the non-initializer-list version of `operator []` can have further C-level array indexing to get an element (or a sub-sub-array if fewer indexing calls are given). This functionality can be accessed from the non-initializer-list versions of `operator ()` and `at()` by giving fewer

indices than `dimensionality`[5]. No version of `[]`, `()`, or `at` accepts more indices than dimensionality; this is checked at compile-time except in the initializer-list version of `at()`, which does it at run-time, and the initializer-list versions of `[]` and `()`, which do *no* length checking.

Due to respecting `dimensionality`, specializations of `array_md<>` without dimension extents do not have the non-initializer-list version of `operator []`. Calling the initializer-list version with no indices, or any version of `operator ()` or `at()` with no indices returns a reference to the entire internal array subobject. This can be done with versions of `array_md<>` with dimensions by calling either `operator ()` or `at()` with no indices.

An important issue is range-checking. Only `at()` does it. If an index is out of range, `at()` throws an `std::out_of_range` exception. Additionally, versions of `at()` taking an initializer list check the number of indices, and throw a `std::length_error` exception if the count is not `dimensionality`. All other functions do *not* check. A range (or length) error results in undefined behavior.

```
array_md<int, 4>     a;     // only four elements!
array_md<int, 3, 2>  b;     // six elements

a[ 5 ] = x;      // RUNTIME ERROR ⇒ undefined behavior
b( 2, 2 ) = y;   // RUNTIME ERROR ⇒ undefined behavior
b( 1, 2 ) = z;   // SOFT ERROR ⇒ undefined behavior but gets b(2,0)
std::cout << a.front();     // OK; unlike std::array, can't be empty
```

So, in doubt you must ensure that the indices for operators `[]` or `()` are valid or use `at()`:

```
template < typename T, unsigned M, unsigned ...N >
void  foo( array_md<T, M, N...> &a )
{
      if ( M > 5 ) {
            a[ 5 ] = …;            // Sort-of OK; not Assignable with non-empty N
            do_something( a[5] ); // Always OK
      }

      a.at( 5 ) = …;   // throws out_of_range exception, if M ≤ 5
                       // compile-time error when sizeof...(N) > 0; since not-Assignable
      a.at( {} ) = …;  // throws length_error exception
      a.at( {5, 0} ) = …;   // throws length_error exception if sizeof...(N) ≠ 1
                            // throws out_of_range exception if M ≤ 5
```

[5] The return type from indexing depends on the number of indices given at compile-time. An initializer list can only given its index count at run-time, so the type is fixed as `value_type` and the count is assumed (or checked in `at()`) to be `dimensionality`.

```
}
```

Note that the number of indices is enforced for you at compile-time, unless you use initializer lists for index groups. The example here from Josuttis' text gave:

```
template < typename C >
void  foo( C &coll ) {
      if ( coll.size() > 5 )
            coll[ 5 ] = …;   // OK
      coll.at( 5 ) = …;      // throws out_of_range exception
}
```

demonstrating dangers using `size()` and `[]` in multi-threaded contexts can't work here because `array_md<>` occasionally works in generic contexts; results from `size()` can't be applied to indexing unless the `array_md<>` has exactly one dimension. The iterator and pointer interfaces can work in generic contexts though.

Iteration

Arrays provide the usual member functions to get iterators: `begin()` and `end()` for forward iteration, in mutable and immutable versions; `cbegin()` and `cend()` to force `const`-mode forward iterations; and `rbegin()`, `rend()`, `crbegin()`, and `crend()` for the reverse iteration versions. These iterators work all the way up to the random-access level. Thus, in principle, you could use all the algorithms of the STL.

They always work in units of `value_type`, no matter what the `dimensionality` is (and not any subarray type). The (forward) iteration order is the same as the lexicographic traversal order used in ordered comparisons. If there are multiple arrays with the same shape, even if the element types differ, and iteration is started with all of them at the same start point (i.e. `begin()`, `rbegin()`, etc.), then updating the iterators in sync will keep them pointing to corresponding elements.

The exact type of the various iterators is implementation-defined. Do not depend on the various forward iterators being C-level pointers; they may be class-types in debugging situations.

Iterators remain valid as long as the array remains valid. However, unlike for all other containers (besides `std::array<>`), `swap()` assigns new values to the elements that iterators, references, and pointers refer to.

Using iterators pushes a linear model on arrays, even when dimensionality is greater than one. An alternate interface to accessing elements can be done with the `apply()` member function. It

takes one argument, a function reference or a similar callable item (function pointer, lambda expression, or class-type with a compatible `operator ()`). The function object is called for each element in the array. The parameters to the function object per call are a reference to an element (a `const` reference when the array is `const`), followed by the index coordinates for the element as `std::size_t` values in "row-major" order. As a basic example:

```
template < typename T, std::size_t R, std::size_t C >
array_md<T, C, R>  transpose( array_md<T, R, C> const &x )
{
      using std::size_t;

      decltype( transpose(x) )  result;

#if 1
      // Either
      result.apply( [&x](T &e, size_t i, size_t j){e = x( j, i );} );
#else
      // Or
      x.apply( [&result](T const &e, size_t i, size_t j)
       {result( j, i ) = e;} );
#endif
}
```

Here, the lambdas have to capture the array that is not calling `apply()` so it can have access. Note that `result`'s capture of `x` could have been by value since `x` was not going to be mutated. The function object's parameters don't have to be exact matches either. The call to `x.apply()` could have taken its first argument by value, since that argument is only being read. Similarly, the following two arguments could have been anything conversion-compatible with `std::size_t`, like some other built-in numeric type. The number of arguments will always be `dimensionality` + 1, so the function object has to support exactly that many parameters, possibly using overloads, default parameters, or variadic arguments:

```
struct element_printer
{
      std::string  make_format( std::size_t count ) {
            if ( count ) {
                  std::string  result = " at (%zu";

                  while ( --count )
                        result += ", %zu";
                  result += ")";
                  return result;
            } else
```

```
                return " as the sole element.";
      }

      template < typename Printable, typename ...Args >
      void  operator ()( Printable const &p, Args const &...args )
      {
            std::cout << '\"' << p << '\"';
            std::printf( make_format(sizeof...( args )).c_str(),
             static_cast<std::size_t>(args)... );
            std::cout << std::endl;
      }
};

int main( int, char *[] ) {
      array_md<int>             a{ 5 };
      array_md<long, 2>         b{ {2L, 7L} };
      array_md<char[6], 2, 2>   c{ {{"Hello", "World"}, {"Video",
       "Watch"}} };
      element_printer           ep;

      a.apply( ep );   // Prints: "5" as the sole element.
      b.apply( ep );   // Like: "2" at (0)□"7" at (1)
      c.apply( ep );   // Like: "Hello" at (0, 0)□"World" at (0, 1)□"Video" at (1, 0)□"Watch"
                       // at (1, 1)
}
```

If calculations are being made during the function call, the code should be path-conservative. The visitation order currently follows forward-iteration order, but your code should not count on that. (That is what the indices being included are for.)

Using `array_md`s as C-Style Arrays

As for classes `std::vector<>` and `std::array<>`, the elements of an `array_md<>` are guaranteed to be in contiguous memory. Thus, you can expect that for an array `a` and nonnegative index `i` that is less than `a.size()`, the following yields `true`:

```
( (T*)&a() )[ i ] == &a.front() + i
```

(Where `T` is the element type.) Josuttis' version had "`&a[i] == &a[0] + i`" above, but that only applies here when the number of dimensions is 1. The contiguity guarantee has some important consequences. It simply means that you can use an `array_md<>` wherever you can

use an ordinary C-style array. For example, you can use an array to hold data of ordinary C-strings of type `char *` or `char const *`:

```
array_md<char, 41>  a;                  // create static array of 41 chars

strcpy( &a[0], "hello, world" );        // copy a C-string into the array
printf( "%s\n", &a[0] );                // print contents of the array as C-string
```

Note that the expression `&a[0]` only works when the number of dimensions is exactly 1. The expression is ill-formed when `dimensionality` is 0, and gives a pointer to an array containing `value_type` when there is more than one dimension (nested when greater than 2). Use the `data()` member function to get the address of the first element as a `value_type *`, which works for any dimension count.

```
array_md<char, 41>   a;                 // create static array of 41 chars
array_md<int, 2, 3>  b{ {5, -4, 0}, {-7, 28, 3} };    // out-of-order data

strcpy( a.data(), "hello, world" );     // copy a C-string into the array
printf( "%s\n", a.data() );             // print contents of the array as C-string
qsort( b.data(), b.size(), sizeof(b.front()),
 [](void const *x, void const *y) noexcept -> int {
     auto const  a = (int const *)x, b = (int const *)y;
     return ( *a == *b ) ? 0 : ( *a < *b ) ? -1 : +1;
 } );                           // capture-less lambda can convert to plain-function pointer
assert( b == decltype(b){-7, -4, 0, 3, 5, 28} );      // confirm the sort
```

The order of elements in memory currently follows that of forward-iteration. Using an `array_md<>` this way brings the same concerns as using C-style arrays and pointers; you must make sure to avoid bounds errors. Plus, when using a character type for the elements, `T{}` has to fill the element after the string data (and there needs to be space for that element). However, these examples show that this type (or `std::vector<>` or `std::array<>`) can be used to provide a C-style array segment with `data()` for use in C-style interfaces.

Note that you must not pass an iterator as the address of the first element. Iterators of class `array_md<>` have an implementation-specific type, which may be totally different from an ordinary pointer:

```
printf( "%s\n", a.begin() );    // ERROR (might work, but not portable)
printf( "%s\n", a.data() );     // OK
```

Exception Handling

Arrays provide only minimal support for logical error checking. The only member function required to throw exceptions is `at()`, which is the safe version of the subscript operator (and of the function-call operator, which is designed to substitute for the subscript operator when the number of indices is not 1).

For functions called by an array (functions for the element type, or functions that are user-supplied, including arguments for `apply()`) no special guarantees are generally given (because you can't insert or delete elements, exceptions might occur only if you copy, move, or assign values). Note especially that `swap()` might throw because it performs element-wise swaps, which may throw.

Tuple Interface

Arrays provide the tuple interface. Thus, you can use the expressions `tuple_size<>::value` to yield the number of elements, `tuple_element<>::type` to yield the type of a specific element, and `get()` to gain access to a specific element. For example:

```
typedef array_md<std::string, 2, 3>  SixStrings;

SixStrings  a = { {"hello", "nico", "how"}, {"are", "you", "?"} };

std::tuple_size<SixStrings>::value          // yields 6
std::tuple_element<1, SixStrings>::type     // yields std::string
get<4>( a )                                 // yields std::string{"you"}
```

Examples of Using Arrays

The following example shows a simple use of class `array_md<>`:

```
#include <boost/container/array_md.hpp>
#include <algorithm>
#include <functional>
#include <iostream>
#include <numeric>
#include <ostream>

template < typename C >
void  print_elements( C const &c )
{
```

```
    for ( auto const &x : c )
        std::cout << x << ' ';
    std::cout << std::endl;
}

int main()
{
    // create one-dimensional array with 10 ints
    boost::container::array_md<int, 10>  a{ 11, 22, 33, 44 };
    print_elements( a );

    // modify last two elements
    a.back() = 9999999;
    a[ a.size() - 2 ] = 42;
    print_elements( a );

    // process sum of all elements
    std::cout << "sum: "
              << std::accumulate( a.begin(), a.end(), 0 )
              << std::endl;

    // negate all elements
    std::transform( a.begin(), a.end(),   // source
     a.begin(),                           // destination
     std::negate<int>{} );                // operation
    print_elements( a );
}
```

As you can see, you can use the general container interface operations (`operator =`, `size()`, and `operator []`) to manipulate the container directly. Because member functions such as `begin()` and `end()` for iterator access are also provided, you can also use different operations that call `begin()` and `end()`, such as modifying and non-modifying algorithms and the auxiliary function `print_elements()`.

The output of the program is as follows:

```
11 22 33 44 0 0 0 0 0 0
11 22 33 44 0 0 0 0 42 9999999
sum: 10000151
-11 -22 -33 -44 0 0 0 0 -42 -9999999
```