Boost.MultiArray Reference Manual

Ronald Garcia, Indiana University

Copyright © 2002 The Trustees of Indiana University

Table of Contents

Library Synopsis 2
MultiArray Concept 3
    Notation 3
    Associated Types 5
    Valid expressions 8
    Complexity guarantees 11
    Invariants 11
    Associated Types for Views 11
    Models 15
Array Components 16
    `multi_array` 17
    `multi_array_ref` 22
    `const_multi_array_ref` 25
Auxiliary Components 28
    `multi_array_types` 28
    `extent_range` 28
    `extent_gen` 29
    Global Objects 29
    View and SubArray Generators 30
    Memory Layout Specifiers 30
    Range Checking 32

Boost.MultiArray is composed of several components. The MultiArray concept defines a generic interface to multidimensional containers. `multi_array` is a general purpose container class that models MultiArray. `multi_array_ref` and `const_multi_array_ref` are adapter classes. Using them, you can manipulate any block of contiguous data as though it were a `multi_array`. `const_multi_array_ref` differs from `multi_array_ref` in that its elements cannot be modified through its interface. Finally, several auxiliary classes are used to create and specialize arrays and some global objects are defined as part of the library interface.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Library Synopsis

To use Boost.MultiArray, you must include the header `boost/multi_array.hpp` in your source. This file brings the following declarations into scope:

```
namespace boost {

  namespace multi_array_types {
    typedef *unspecified* index;
    typedef *unspecified* size_type;
    typedef *unspecified* difference_type;
    typedef *unspecified* index_range;
    typedef *unspecified* extent_range;
    typedef *unspecified* index_gen;
    typedef *unspecified* extent_gen;
  }

  template <typename ValueType,
            std::size_t NumDims,
            typename Allocator = std::allocator<ValueType> >
  class multi_array;

  template <typename ValueType,
            std::size_t NumDims>
  class multi_array_ref;

  template <typename ValueType,
            std::size_t NumDims>
  class const_multi_array_ref;

  multi_array_types::extent_gen extents;
  multi_array_types::index_gen  indices;

  template <typename Array, int N> class subarray_gen;
  template <typename Array, int N> class const_subarray_gen;
  template <typename Array, int N> class array_view_gen;
  template <typename Array, int N> class const_array_view_gen;

  class c_storage_order;
  class fortran_storage_order;
  template <std::size_t NumDims> class general_storage_order;

}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

MultiArray Concept

The MultiArray concept defines an interface to hierarchically nested containers. It specifies operations for accessing elements, traversing containers, and creating views of array data. MultiArray defines a flexible memory model that accomodates a variety of data layouts.

At each level (or dimension) of a MultiArray's container hierarchy lie a set of ordered containers, each of which contains the same number and type of values. The depth of this container hierarchy is the MultiArray's *dimensionality*. MultiArray is recursively defined; the containers at each level of the container hierarchy model MultiArray as well. While each dimension of a MultiArray has its own size, the list of sizes for all dimensions defines the *shape* of the entire MultiArray. At the base of this hierarchy lie 1-dimensional MultiArrays. Their values are the contained objects of interest and not part of the container hierarchy. These are the MultiArray's elements.

Like other container concepts, MultiArray exports iterators to traverse its values. In addition, values can be addressed directly using the familiar bracket notation.

MultiArray also specifies routines for creating specialized views. A *view* lets you treat a subset of the underlying elements in a MultiArray as though it were a separate MultiArray. Since a view refers to the same underlying elements, changes made to a view's elements will be reflected in the original MultiArray. For example, given a 3-dimensional "cube" of elements, a 2-dimensional slice can be viewed as if it were an independent MultiArray. Views are created using `index_gen` and `index_range` objects. `index_ranges` denote elements from a certain dimension that are to be included in a view. `index_gen` aggregates range data and performs bookkeeping to determine the view type to be returned. MultiArray's `operator[]` must be passed the result of `N` chained calls to `index_gen::operator[]`, i.e.

```
indices[a0][a1]...[aN];
```

where `N` is the MultiArray's dimensionality and `indices` an object of type `index_gen`. The view type is dependent upon the number of degenerate dimensions specified to `index_gen`. A degenerate dimension occurs when a single-index is specified to `index_gen` for a certain dimension. For example, if `indices` is an object of type `index_gen`, then the following example:

```
indices[index_range(0,5)][2][index_range(0,4)];
```

has a degenerate second dimension. The view generated from the above specification will have 2 dimensions with shape `5 x 4`. If the "`2`" above were replaced with another `index_range` object, for example:

```
indices[index_range(0,5)][index_range(0,2)][index_range(0,4)];
```

then the view would have 3 dimensions.

MultiArray exports information regarding the memory layout of its contained elements. Its memory model for elements is completely defined by 4 properties: the origin, shape, index bases, and strides. The origin is the address in memory of the element accessed as `a[0][0]...[0]`, where `a` is a MultiArray. The shape is a list of numbers specifying the size of containers at each dimension. For example, the first extent is the size of the outermost container, the second extent is the size of its subcontainers, and so on. The index bases are a list of signed values specifying the index of the first value in a container. All containers at the same dimension share the same index base. Note that since positive index bases are possible, the origin need not exist in order to determine the location in memory of the MultiArray's elements. The strides determine how index values are mapped to memory offsets. They accomodate a number of possible element layouts. For example, the elements of a 2 dimensional array can be stored by row (i.e., the elements of each row are stored contiguously) or by column (i.e., the elements of each column are stored contiguously).

Two concept checking classes for the MultiArray concepts (`ConstMultiArrayConcept` and `MutableMultiArrayConcept`) are in the namespace `boost::multi_array_concepts` in `<boost/multi_array/concept_checks.hpp>`.

Notation

What follows are the descriptions of symbols that will be used to describe the MultiArray interface.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 1. Notation

`A`	A type that is a model of MultiArray
`a,b`	Objects of type `A`
`NumDims`	The numeric dimension parameter associated with `A`.
`Dims`	Some numeric dimension parameter such that `0<Dims<NumDims`.
`indices`	An object created by some number of chained calls to `index_gen::operator[](index_range)`.
`index_list`	An object whose type models Collection
`idx`	A signed integral value.
`tmp`	An object of type `boost::array<index,NumDims>`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Associated Types

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 2. Associated Types

Type	Description
`value_type`	This is the value type of the container. If `NumDims == 1`, then this is `element`. Otherwise, this is the value type of the immediately nested containers.
`reference`	This is the reference type of the contained value. If `NumDims == 1`, then this is `element&`. Otherwise, this is the same type as `template subarray<NumDims-1>::type`.
`const_reference`	This is the const reference type of the contained value. If `NumDims == 1`, then this is `const element&`. Otherwise, this is the same type as `template const_subarray<NumDims-1>::type`.
`size_type`	This is an unsigned integral type. It is primarily used to specify array shape.
`difference_type`	This is a signed integral type used to represent the distance between two iterators. It is the same type as `std::iterator_traits<iterator>::difference_type`.
`iterator`	This is an iterator over the values of `A`. If `NumDims == 1`, then it models `Random Access Iterator`. Otherwise it models Random Access Traversal Iterator, Readable Iterator, Writable Iterator, and `Output Iterator`.
`const_iterator`	This is the const iterator over the values of `A`.
`reverse_iterator`	This is the reversed iterator, used to iterate backwards over the values of `A`.
`const_reverse_iterator`	This is the reversed const iterator. `A`.
`element`	This is the type of objects stored at the base of the hierarchy of MultiArrays. It is the same as `template subarray<1>::value_type`
`index`	This is a signed integral type used for indexing into `A`. It is also used to represent strides and index bases.
`index_gen`	This type is used to create a tuple of `index_ranges` passed to `operator[]` to create an `array_view<Dims>::type` object.
`index_range`	This type specifies a range of indices over some dimension of a MultiArray. This range will be visible through an `array_view<Dims>::type` object.
`template subarray<Dims>::type`	This is subarray type with `Dims` dimensions. It is the reference type of the `(NumDims - Dims)` dimension of `A` and also models MultiArray.
`template const_subarray<Dims>::type`	This is the const subarray type.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Type	**Description**
`template array_view<Dims>::type`	This is the view type with `Dims` dimensions. It is returned by calling `operator[](indices)`. It models MultiArray.
`template const_array_view<Dims>::type`	This is the const view type with `Dims` dimensions.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Valid expressions

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 3. Valid Expressions

Expression	**Return type**	**Semantics**
`A::dimensionality`	`size_type`	This compile-time constant represents the number of dimensions of the array (note that `A::dimensionality == NumDims`).
`a.shape()`	`const size_type*`	This returns a list of `NumDims` elements specifying the extent of each array dimension.
`a.strides()`	`const index*`	This returns a list of `NumDims` elements specifying the stride associated with each array dimension. When accessing values, strides is used to calculate an element's location in memory.
`a.index_bases()`	`const index*`	This returns a list of `NumDims` elements specifying the numeric index of the first element for each array dimension.
`a.origin()`	`element*` if a is mutable, `const element*` otherwise.	This returns the address of the element accessed by the expression `a[0][0]...[0].`. If the index bases are positive, this element won't exist, but the address can still be used to locate a valid element given its indices.
`a.num_dimensions()`	`size_type`	This returns the number of dimensions of the array (note that `a.num_dimensions() == NumDims`).
`a.num_elements()`	`size_type`	This returns the number of elements contained in the array. It is equivalent to the following code:

```
std::accumu↲
late(a.shape(),a.shape+a.num_di↲
mensions(),
    size_type(1),std::multi↲
plies<size_type>());
```

`a.size()`	`size_type`	This returns the number of values contained in a. It is equivalent to `a.shape()[0];`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Expression	Return type	Semantics
`a(index_list)`	`element&`; if `a` is mutable, `const element&` otherwise.	This expression accesses a specific element of `a`.`index_list` is the unique set of indices that address the element returned. It is equivalent to the following code (disregarding intermediate temporaries):

```

    // multiply indices by strides
    std::transform(index_list.begin(), index_list.end(),
      a.strides(), tmp.begin(), std::multiplies<index>()),

    // add the sum of the products to the origin
    *std::accumulate(tmp.begin(), tmp.end(), a.origin());
```

Expression	Return type	Semantics
`a.begin()`	`iterator` if `a` is mutable, `const_iterator` otherwise.	This returns an iterator pointing to the beginning of `a`.
`a.end()`	`iterator` if `a` is mutable, `const_iterator` otherwise.	This returns an iterator pointing to the end of `a`.
`a.rbegin()`	`reverse_iterator` if `a` is mutable, `const_reverse_iterator` otherwise.	This returns a reverse iterator pointing to the beginning of `a` reversed.
`a.rend()`	`reverse_iterator` if `a` is mutable, `const_reverse_iterator` otherwise.	This returns a reverse iterator pointing to the end of `a` reversed.
`a[idx]`	`reference` if `a` is mutable, `const_reference` otherwise.	This returns a reference type that is bound to the index `idx` value of `a`. Note that if `i` is the index base for this dimension, the above expression returns the `(idx-i)`th element (counting from zero). The expression is equivalent to `*(a.begin()+idx-a.index_bases()[0]);`.
`a[indices]`	`array_view<Dims>::type` if `a` is mutable, `const_array_view<Dims>::type` otherwise.	This expression generates a view of the array determined by the `index_range` and `index` values used to construct `indices`.
`a == b`	bool	This performs a lexicographical comparison of the values of `a` and `b`. The element type must model EqualityComparable for this expression to be valid.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Expression	**Return type**	**Semantics**
`a < b`	bool	This performs a lexicographical comparison of the values of `a` and `b`. The element type must model LessThanComparable for this expression to be valid.
`a <= b`	bool	This performs a lexicographical comparison of the values of `a` and `b`. The element type must model EqualityComparable and LessThanComparable for this expression to be valid.
`a > b`	bool	This performs a lexicographical comparison of the values of `a` and `b`. The element type must model EqualityComparable and LessThanComparable for this expression to be valid.
`a >= b`	bool	This performs a lexicographical comparison of the values of `a` and `b`. The element type must model LessThanComparable for this expression to be valid.

Complexity guarantees

`begin()` and `end()` execute in amortized constant time. `size()` executes in at most linear time in the MultiArray's size.

Invariants

Table 4. Invariants

Valid range	`[a.begin(),a.end())` is a valid range.
Range size	`a.size() == std::distance(a.begin(),a.end());`.
Completeness	Iteration through the range `[a.begin(),a.end())` will traverse across every `value_type` of `a`.
Accessor Equivalence	Calling `a[a1][a2]...[aN]` where `N==NumDims` yields the same result as calling `a(index_list)`, where `index_list` is a Collection containing the values `a1...aN`.

Associated Types for Views

The following MultiArray associated types define the interface for creating views of existing MultiArrays. Their interfaces and roles in the concept are described below.

`index_range`

`index_range` objects represent half-open strided intervals. They are aggregated (using an `index_gen` object) and passed to a MultiArray's `operator[]` to create an array view. When creating a view, each `index_range` denotes a range of valid indices along one dimension of a MultiArray. Elements that are accessed through the set of ranges specified will be included in the constructed view. In some cases, an `index_range` is created without specifying start or finish values. In those cases, the object is interpreted to start at the beginning of a MultiArray dimension and end at its end.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`index_range` objects can be constructed and modified several ways in order to allow convenient and clear expression of a range of indices. To specify ranges, `index_range` supports a set of constructors, mutating member functions, and a novel specification involving inequality operators. Using inequality operators, a half open range [5,10) can be specified as follows:

```
5 <= index_range() < 10;
```

or

```
4 < index_range() <= 9;
```

and so on. The following describes the `index_range` interface.

Table 5. Notation

`i`	An object of type `index_range`.
`idx,idx1,idx2,idx3`	Objects of type `index`.

Table 6. Associated Types

Type	**Description**
`index`	This is a signed integral type. It is used to specify the start, finish, and stride values.
`size_type`	This is an unsigned integral type. It is used to report the size of the range an `index_range` represents.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 7. Valid Expressions

Expression	Return type	Semantics
`index_range(idx1,idx2,idx3)`	`index_range`	This constructs an `index_range` representing the interval `[idx1,idx2)` with stride `idx3`.
`index_range(idx1,idx2)`	`index_range`	This constructs an `index_range` representing the interval `[idx1,idx2)` with unit stride. It is equivalent to `index_range(idx1,idx2,1)`.
`index_range()`	`index_range`	This construct an `index_range` with unspecified start and finish values.
`i.start(idx1)`	`index&`	This sets the start index of `i` to `idx`.
`i.finish(idx)`	`index&`	This sets the finish index of `i` to `idx`.
`i.stride(idx)`	`index&`	This sets the stride length of `i` to `idx`.
`i.start()`	`index`	This returns the start index of `i`.
`i.finish()`	`index`	This returns the finish index of `i`.
`i.stride()`	`index`	This returns the stride length of `i`.
`i.get_start(idx)`	`index`	If `i` specifies a start value, this is equivalent to `i.start()`. Otherwise it returns `idx`.
`i.get_finish(idx)`	`index`	If `i` specifies a finish value, this is equivalent to `i.finish()`. Otherwise it returns `idx`.
`i.size(idx)`	`size_type`	If `i` specifies a both finish and start values, this is equivalent to `(i.finish()-i.start())/i.stride()`. Otherwise it returns `idx`.
`i < idx`	`index`	This is another syntax for specifying the finish value. This notation does not include `idx` in the range of valid indices. It is equivalent to `index_range(r.start(), idx, r.stride())`
`i <= idx`	`index`	This is another syntax for specifying the finish value. This notation includes `idx` in the range of valid indices. It is equivalent to `index_range(r.start(), idx + 1, r.stride())`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Expression	**Return type**	**Semantics**
`idx < i`	`index`	This is another syntax for specifying the start value. This notation does not include `idx` in the range of valid indices. It is equivalent to `index_range(idx + 1, i.finish(), i.stride())`.
`idx <= i`	`index`	This is another syntax for specifying the start value. This notation includes `idx1` in the range of valid indices. It is equivalent to `index_range(idx, i.finish(), i.stride())`.
`i + idx`	`index`	This expression shifts the start and finish values of `i` up by `idx`. It is equivalent to `index_range(r.start()+idx1, r.finish()+idx, r.stride())`
`i - idx`	`index`	This expression shifts the start and finish values of `i` up by `idx`. It is equivalent to `index_range(r.start()-idx1, r.finish()-idx, r.stride())`

`index_gen`

`index_gen` aggregates `index_range` objects in order to specify view parameters. Chained calls to `operator[]` store range and dimension information used to instantiate a new view into a MultiArray.

Table 8. Notation

`Dims,Ranges`	Unsigned integral values.
`x`	An object of type `template gen_type<Dims,Ranges>::type`.
`i`	An object of type `index_range`.
`idx`	Objects of type `index`.

Table 9. Associated Types

Type	**Description**
`index`	This is a signed integral type. It is used to specify degenerate dimensions.
`size_type`	This is an unsigned integral type. It is used to report the size of the range an `index_range` represents.
`template gen_type::<Dims,Ranges>::type`	This type generator names the result of `Dims` chained calls to `index_gen::operator[]`. The `Ranges` parameter is determined by the number of degenerate ranges specified (i.e. calls to `operator[](index)`). Note that `index_gen` and `gen_type<0,0>::type` are the same type.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 10. Valid Expressions

Expression	**Return type**	**Semantics**
`index_gen()`	`gen_type<0,0>::type`	This constructs an `index_gen` object. This object can then be used to generate tuples of `index_range` values.
`x[i]`	`gen_type<Dims+1,Ranges+1>::type`	Returns a new object containing all previous `index_range` objects in addition to `i.` Chained calls to `operator[]` are the means by which `index_range` objects are aggregated.
`x[idx]`	`gen_type<Dims,Ranges+1>::type`	Returns a new object containing all previous `index_range` objects in addition to a degenerate range, `index_range(idx,idx).` Note that this is NOT equivalent to `x[index_range(idx,idx)].`, which will return an object of type `gen_type<Dims+1,Ranges+1>::type`.

Models

- `multi_array`
- `multi_array_ref`
- `const_multi_array_ref`
- `template array_view<Dims>::type`
- `template const_array_view<Dims>::type`
- `template subarray<Dims>::type`
- `template const_subarray<Dims>::type`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Array Components

Boost.MultiArray defines an array class, `multi_array`, and two adapter classes, `multi_array_ref` and `const_multi_array_ref`. The three classes model MultiArray and so they share a lot of functionality. `multi_array_ref` differs from `multi_array` in that the `multi_array` manages its own memory, while `multi_array_ref` is passed a block of memory that it expects to be externally managed. `const_multi_array_ref` differs from `multi_array_ref` in that the underlying elements it adapts cannot be modified through its interface, though some array properties, including the array shape and index bases, can be altered. Functionality the classes have in common is described below.

Note: Preconditions, Effects, and Implementation. Throughout the following sections, small pieces of C++ code are used to specify constraints such as preconditions, effects, and postconditions. These do not necessarily describe the underlying implementation of array components; rather, they describe the expected input to and behavior of the specified operations. Failure to meet preconditions results in undefined behavior. Not all effects (i.e. copy constructors, etc.) must be mimicked exactly. The code snippets for effects intend to capture the essence of the described operation.

Queries.

```
element* data();
const element* data() ↵
const;
```

This returns a pointer to the beginning of the contiguous block that contains the array's data. If all dimensions of the array are 0-indexed and stored in ascending order, this is equivalent to `origin()`. Note that `const_multi_array_ref` only provides the const version of this function.

```
element* origin();
const element* origin() ↵
const;
```

This returns the origin element of the `multi_array`. Note that `const_multi_array_ref` only provides the const version of this function. (Required by MultiArray)

`const index* index_bases();` This returns the index bases for the `multi_array`. (Required by MultiArray)

`const index* strides();` This returns the strides for the `multi_array`. (Required by MultiArray)

`const size_type* shape();` This returns the shape of the `multi_array`. (Required by MultiArray)

Comparators.

```
bool operator==(const *ar↵
ray-type*& rhs);
bool operator!=(const *ar↵
ray-type*& rhs);
bool operator<(const *ar↵
ray-type*& rhs);
bool operator>(const *ar↵
ray-type*& rhs);
bool operator>=(const *ar↵
ray-type*& rhs);
bool operator<=(const *ar↵
ray-type*& rhs);
```

Each comparator executes a lexicographical compare over the value types of the two arrays. (Required by MultiArray)

Preconditions. `element` must support the comparator corresponding to that called on `multi_array`.

Complexity. O(`num_elements()`).

Modifiers.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
template <typename SizeL↲
ist>
void reshape(const SizeL↲
ist& sizes)
```

This changes the shape of the `multi_array`. The number of elements and the index bases remain the same, but the number of values at each level of the nested container hierarchy may change.

`SizeList` Requirements. `SizeList` must model Collection.

Preconditions.

```
std::accumulate(sizes.be↲
gin(),sizes.end(),size_type(1),std::times<size_type>()) == this-
>num_elements();
sizes.size() == NumDims;
```

Postconditions. `std::equal(sizes.begin(),sizes.end(),this->shape) == true;`

```
template <typename BaseL↲
ist>
void reindex(const BaseL↲
ist& values);
```

This changes the index bases of the `multi_array` to correspond to the the values in `values`.

`BaseList` Requirements. `BaseList` must model Collection.

Preconditions. `values.size() == NumDims;`

Postconditions. `std::equal(values.begin(),values.end(),this->index_bases());`

```
void reindex(index ↲
value);
```

This changes the index bases of all dimensions of the `multi_array` to `value`.

Postconditions.

```
std::count_if(this->index_bases(),this->index_bases()+this->num_di↲
mensions(),
              std::bind_2nd(std::equal_to<index>(),value)) ==
              this->num_dimensions();
```

`multi_array`

`multi_array` is a multi-dimensional container that supports random access iteration. Its number of dimensions is fixed at compile time, but its shape and the number of elements it contains are specified during its construction. The number of elements will remain fixed for the duration of a `multi_array`'s lifetime, but the shape of the container can be changed. A `multi_array` manages its data elements using a replaceable allocator.

Model Of. MultiArray, CopyConstructible. Depending on the element type, it may also model EqualityComparable and LessThanComparable.

Synopsis.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace boost {

template <typename ValueType, 
          std::size_t NumDims, 
          typename Allocator = std::allocator<ValueType> >
class multi_array {
public:
// types:
  typedef ValueType                             element;
  typedef *unspecified*                         value_type;
  typedef *unspecified*                         reference;
  typedef *unspecified*                         const_reference;
  typedef *unspecified*                         difference_type;
  typedef *unspecified*                         iterator;
  typedef *unspecified*                         const_iterator;
  typedef *unspecified*                         reverse_iterator;
  typedef *unspecified*                         const_reverse_iterator;
  typedef multi_array_types::size_type          size_type;
  typedef multi_array_types::index              index;
  typedef multi_array_types::index_gen          index_gen;
  typedef multi_array_types::index_range        index_range;
  typedef multi_array_types::extent_gen         extent_gen;
  typedef multi_array_types::extent_range       extent_range;
  typedef *unspecified*                         storage_order_type;


  // template typedefs
  template <std::size_t Dims> struct            subarray;
  template <std::size_t Dims> struct            const_subarray;
  template <std::size_t Dims> struct            array_view;
  template <std::size_t Dims> struct            const_array_view;


  static const std::size_t dimensionality = NumDims;


  // constructors and destructors

  multi_array();

  template <typename ExtentList>
  explicit multi_array(const ExtentList& sizes,
                       const storage_order_type& store = c_storage_order(),
                       const Allocator& alloc = Allocator());
  explicit multi_array(const extents_tuple& ranges,
                       const storage_order_type& store = c_storage_order(),
                const Allocator& alloc = Allocator());
  multi_array(const multi_array& x);
  multi_array(const const_multi_array_ref<ValueType,NumDims>& x);
  multi_array(const const_subarray<NumDims>::type& x);
  multi_array(const const_array_view<NumDims>::type& x);

  multi_array(const multi_array_ref<ValueType,NumDims>& x);
  multi_array(const subarray<NumDims>::type& x);
  multi_array(const array_view<NumDims>::type& x);

  ~multi_array();

  // modifiers

  multi_array& operator=(const multi_array& x);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  template <class Array> multi_array& operator=(const Array& x);

  // iterators:
  iterator    begin();
  iterator    end();
  const_iterator   begin() const;
  const_iterator   end() const;
  reverse_iterator   rbegin();
  reverse_iterator   rend();
  const_reverse_iterator  rbegin() const;
  const_reverse_iterator  rend() const;

  // capacity:
  size_type    size() const;
  size_type    num_elements() const;
  size_type    num_dimensions() const;

  // element access:
  template <typename IndexList>
    element&   operator()(const IndexList& indices);
  template <typename IndexList>
    const element&  operator()(const IndexList& indices) const;
  reference   operator[](index i);
  const_reference  operator[](index i) const;
  array_view<Dims>::type operator[](const indices_tuple& r);
  const_array_view<Dims>::type operator[](const indices_tuple& r) const;

  // queries
  element*   data();
  const element*  data() const;
  element*   origin();
  const element*  origin() const;
  const size_type*  shape() const;
  const index*   strides() const;
  const index*   index_bases() const;
  const storage_order_type&     storage_order() const;

  // comparators
  bool operator==(const multi_array& rhs);
  bool operator!=(const multi_array& rhs);
  bool operator<(const multi_array& rhs);
  bool operator>(const multi_array& rhs);
  bool operator>=(const multi_array& rhs);
  bool operator<=(const multi_array& rhs);

  // modifiers:
  template <typename InputIterator>
    void   assign(InputIterator begin, InputIterator end);
  template <typename SizeList>
    void   reshape(const SizeList& sizes)
  template <typename BaseList> void reindex(const BaseList& values);
    void   reindex(index value);
  template <typename ExtentList>
    multi_array&  resize(const ExtentList& extents);
  multi_array&                  resize(extents_tuple& extents);
};
```

Constructors.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
template <typename Extent↲
List>
explicit multi_ar↲
ray(const ExtentList& ↲
sizes,
                     ↲
const storage_or↲
der_type& store = c_stor↲
age_order(),
                     ↲
const Allocator& alloc = ↲
Allocator());
```

This constructs a `multi_array` using the specified parameters. `sizes` specifies the shape of the constructed `multi_array`. `store` specifies the storage order or layout in memory of the array dimensions. `alloc` is used to allocate the contained elements.

`ExtentList` Requirements. `ExtentList` must model Collection.

Preconditions. `sizes.size() == NumDims;`

```
explicit multi_array(ex↲
tent_gen::gen_type<Num↲
Dims>::type ranges,
                     ↲
const storage_or↲
der_type& store = c_stor↲
age_order(),
                     ↲
const Allocator& alloc = ↲
Allocator());
```

This constructs a `multi_array` using the specified parameters. `ranges` specifies the shape and index bases of the constructed multi_array. It is the result of `NumDims` chained calls to `extent_gen::operator[]`. `store` specifies the storage order or layout in memory of the array dimensions. `alloc` is the allocator used to allocate the memory used to store `multi_array` elements.

```

multi_array(const ↲
multi_array& x);
multi_array(const ↲
const_multi_ar↲
ray_ref<ValueType,Num↲
Dims>& x);
multi_array(const ↲
const_subarray<Num↲
Dims>::type& x);
multi_array(const ↲
const_array_view<Num↲
Dims>::type& x);
multi_array(const ↲
multi_ar↲
ray_ref<ValueType,Num↲
Dims>& x);
multi_array(const subar↲
ray<NumDims>::type& x);
multi_array(const ar↲
ray_view<NumDims>::type& ↲
x);
```

These constructors all constructs a `multi_array` and perform a deep copy of `x`.

Complexity. This performs O(`x.num_elements()`) calls to `element`'s copy constructor.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
multi_array();
```

This constructs a `multi_array` whose shape is (0,...,0) and contains no elements.

Note on Constructors. The `multi_array` construction expressions,

```
    multi_array<int,3> A(boost::extents[5][4][3]);
```

and

```
    boost::array<multi_array_base::index,3> my_extents = {{5, 4, 3}};
    multi_array<int,3> A(my_extents);
```

are equivalent.

Modifiers.

```
multi_array& operat↲
or=(const multi_array& ↲
x);
template <class Array> ↲
multi_array& operat↲
or=(const Array& x);
```

This performs an element-wise copy of `x` into the current `multi_array`.

`Array` Requirements. `Array` must model MultiArray.

Preconditions.

```
std::equal(this->shape(),this->shape()+this->num_dimensions(),
x.shape());
```

Postconditions.

```
(*.this) == x;
```

Complexity. The assignment operators perform O(`x.num_elements()`) calls to `element`'s copy constructor.

```
template <typename In↲
putIterator>
void assign(InputIterat↲
or begin, InputIterator ↲
end);
```

This copies the elements in the range `[begin,end)` into the array. It is equivalent to `std::copy(begin,end,this->data())`.

Preconditions. `std::distance(begin,end) == this->num_elements();`

Complexity. The `assign` member function performs O(`this->num_elements()`) calls to `ValueType`'s copy constructor.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
multi_array& resize(ex↲
tent_gen::gen_type<Num↲
Dims>::type extents);
template <typename Extent↲
List>
  multi_array& res↲
ize(const ExtentList& ex↲
tents);
```

This function resizes an array to the shape specified by `extents`, which is either a generated list of extents or a model of the `Collection` concept. The contents of the array are preserved whenever possible; if the new array size is smaller, then some data will be lost. Any new elements created by resizing the array are initialized with the `element` default constructor.

Queries.

```
storage_order_type& stor↲
age_order() const;
```

This query returns the storage order object associated with the `multi_array` in question. It can be used to construct a new array with the same storage order.

`multi_array_ref`

`multi_array_ref` is a multi-dimensional container adaptor. It provides the MultiArray interface over any contiguous block of elements. `multi_array_ref` exports the same interface as `multi_array`, with the exception of the constructors.

Model Of. `multi_array_ref` models MultiArray, CopyConstructible. and depending on the element type, it may also model EqualityComparable and LessThanComparable. Detailed descriptions are provided here only for operations that are not described in the `multi_array` reference.

Synopsis.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace boost {

template <typename ValueType,
          std::size_t NumDims>
class multi_array_ref {
public:
// types:
  typedef ValueType                             element;
  typedef *unspecified*                         value_type;
  typedef *unspecified*                         reference;
  typedef *unspecified*                         const_reference;
  typedef *unspecified*                         difference_type;
  typedef *unspecified*                         iterator;
  typedef *unspecified*                         const_iterator;
  typedef *unspecified*                         reverse_iterator;
  typedef *unspecified*                         const_reverse_iterator;
  typedef multi_array_types::size_type          size_type;
  typedef multi_array_types::index              index;
  typedef multi_array_types::index_gen          index_gen;
  typedef multi_array_types::index_range        index_range;
  typedef multi_array_types::extent_gen         extent_gen;
  typedef multi_array_types::extent_range       extent_range;
  typedef *unspecified*                         storage_order_type;

  // template typedefs
  template <std::size_t Dims> struct            subarray;
  template <std::size_t Dims> struct            const_subarray;
  template <std::size_t Dims> struct            array_view;
  template <std::size_t Dims> struct            const_array_view;


  static const std::size_t dimensionality = NumDims;


  // constructors and destructors

  template <typename ExtentList>
  explicit multi_array_ref(element* data, const ExtentList& sizes,
                     const storage_order_type& store = c_storage_order());
  explicit multi_array_ref(element* data, const extents_tuple& ranges,
                     const storage_order_type& store = c_storage_order());
  multi_array_ref(const multi_array_ref& x);
  ~multi_array_ref();

  // modifiers

  multi_array_ref& operator=(const multi_array_ref& x);
  template <class Array> multi_array_ref& operator=(const Array& x);

  // iterators:
  iterator    begin();
  iterator    end();
  const_iterator   begin() const;
  const_iterator   end() const;
  reverse_iterator   rbegin();
  reverse_iterator   rend();
  const_reverse_iterator  rbegin() const;
  const_reverse_iterator  rend() const;

  // capacity:
  size_type    size() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  size_type    num_elements() const;
  size_type    num_dimensions() const;

  // element access:
  template <typename IndexList>
    element&   operator()(const IndexList& indices);
  template <typename IndexList>
    const element&  operator()(const IndexList& indices) const;
  reference   operator[](index i);
  const_reference  operator[](index i) const;
  array_view<Dims>::type operator[](const indices_tuple& r);
  const_array_view<Dims>::type operator[](const indices_tuple& r) const;

  // queries
  element*   data();
  const element*  data() const;
  element*   origin();
  const element*  origin() const;
  const size_type*  shape() const;
  const index*   strides() const;
  const index*   index_bases() const;
  const storage_order_type&     storage_order() const;

  // comparators
  bool operator==(const multi_array_ref& rhs);
  bool operator!=(const multi_array_ref& rhs);
  bool operator<(const multi_array_ref& rhs);
  bool operator>(const multi_array_ref& rhs);
  bool operator>=(const multi_array_ref& rhs);
  bool operator<=(const multi_array_ref& rhs);

  // modifiers:
  template <typename InputIterator>
    void   assign(InputIterator begin, InputIterator end);
  template <typename SizeList>
    void   reshape(const SizeList& sizes)
  template <typename BaseList> void reindex(const BaseList& values);
  void    reindex(index value);
};
```

Constructors.

```
template <typename Extent↵
List>
explicit multi_ar↵
ray_ref(element* data,
                     ↵
const ExtentList& sizes,
                     ↵
const storage_order& ↵
store = c_storage_or↵
der(),
                     ↵
const Allocator& alloc = ↵
Allocator());
```

This constructs a `multi_array_ref` using the specified parameters. `sizes` specifies the shape of the constructed `multi_array_ref`. `store` specifies the storage order or layout in memory of the array dimensions. `alloc` is used to allocate the contained elements.

`ExtentList` Requirements. `ExtentList` must model Collection.

Preconditions. `sizes.size() == NumDims;`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
explicit multi_ar↵
ray_ref(element* data,
                     ex↵
tent_gen::gen_type<Num↵
Dims>::type ranges,
                      ↵
const storage_order& ↵
store = c_storage_or↵
der());
```

This constructs a `multi_array_ref` using the specified parameters. `ranges` specifies the shape and index bases of the constructed multi_array_ref. It is the result of `NumDims` chained calls to `extent_gen::operator[]`. `store` specifies the storage order or layout in memory of the array dimensions.

```
multi_array_ref(const ↵
multi_array_ref& x);
```

This constructs a shallow copy of `x`.

Complexity. Constant time (for contrast, compare this to the `multi_array` class copy constructor.

Modifiers.

```
multi_array_ref& operat↵
or=(const multi_ar↵
ray_ref& x);
template <class Array> ↵
multi_array_ref& operat↵
or=(const Array& x);
```

This performs an element-wise copy of `x` into the current `multi_array_ref`.

`Array` Requirements. `Array` must model MultiArray.

Preconditions.

```
std::equal(this->shape(),this->shape()+this->num_dimensions(),
x.shape());
```

Postconditions.

```
(*.this) == x;
```

Complexity. The assignment operators perform O(`x.num_elements()`) calls to `element`'s copy constructor.

`const_multi_array_ref`

`const_multi_array_ref` is a multi-dimensional container adaptor. It provides the MultiArray interface over any contiguous block of elements. `const_multi_array_ref` exports the same interface as `multi_array`, with the exception of the constructors.

Model Of. `const_multi_array_ref` models MultiArray, CopyConstructible. and depending on the element type, it may also model EqualityComparable and LessThanComparable. Detailed descriptions are provided here only for operations that are not described in the `multi_array` reference.

Synopsis.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace boost {

template <typename ValueType,
          std::size_t NumDims,
          typename TPtr = const T*>
class const_multi_array_ref {
public:
// types:
  typedef ValueType                             element;
  typedef *unspecified*                         value_type;
  typedef *unspecified*                         reference;
  typedef *unspecified*                         const_reference;
  typedef *unspecified*                         difference_type;
  typedef *unspecified*                         iterator;
  typedef *unspecified*                         const_iterator;
  typedef *unspecified*                         reverse_iterator;
  typedef *unspecified*                         const_reverse_iterator;
  typedef multi_array_types::size_type          size_type;
  typedef multi_array_types::index              index;
  typedef multi_array_types::index_gen          index_gen;
  typedef multi_array_types::index_range        index_range;
  typedef multi_array_types::extent_gen         extent_gen;
  typedef multi_array_types::extent_range       extent_range;
  typedef *unspecified*                         storage_order_type;

  // template typedefs
  template <std::size_t Dims> struct            subarray;
  template <std::size_t Dims> struct            const_subarray;
  template <std::size_t Dims> struct            array_view;
  template <std::size_t Dims> struct            const_array_view;


  // structors

  template <typename ExtentList>
  explicit const_multi_array_ref(TPtr data, const ExtentList& sizes,
                       const storage_order_type& store = c_storage_order());
  explicit const_multi_array_ref(TPtr data, const extents_tuple& ranges,
                       const storage_order_type& store = c_storage_order());
  const_multi_array_ref(const const_multi_array_ref& x);
  ~const_multi_array_ref();



  // iterators:
  const_iterator   begin() const;
  const_iterator   end() const;
  const_reverse_iterator  rbegin() const;
  const_reverse_iterator  rend() const;

  // capacity:
  size_type    size() const;
  size_type    num_elements() const;
  size_type    num_dimensions() const;

  // element access:
  template <typename IndexList>
    const element&  operator()(const IndexList& indices) const;
  const_reference  operator[](index i) const;
  const_array_view<Dims>::type operator[](const indices_tuple& r) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  // queries
  const element*  data() const;
  const element*  origin() const;
  const size_type*  shape() const;
  const index*   strides() const;
  const index*   index_bases() const;
  const storage_order_type&     storage_order() const;

  // comparators
  bool operator==(const const_multi_array_ref& rhs);
  bool operator!=(const const_multi_array_ref& rhs);
  bool operator<(const const_multi_array_ref& rhs);
  bool operator>(const const_multi_array_ref& rhs);
  bool operator>=(const const_multi_array_ref& rhs);
  bool operator<=(const const_multi_array_ref& rhs);

  // modifiers:
  template <typename SizeList>
  void   reshape(const SizeList& sizes)
  template <typename BaseList> void reindex(const BaseList& values);
  void    reindex(index value);
};
```

Constructors.

```
template <typename Extent↲
List>
explicit const_multi_ar↲
ray_ref(TPtr data,
                       ↲
const ExtentList& sizes,
                       ↲
const storage_order& ↲
store = c_storage_or↲
der());
```

This constructs a `const_multi_array_ref` using the specified parameters. `sizes` specifies the shape of the constructed `const_multi_array_ref`. `store` specifies the storage order or layout in memory of the array dimensions.

`ExtentList` Requirements. `ExtentList` must model Collection.

Preconditions. `sizes.size() == NumDims;`

```
explicit const_multi_ar↲
ray_ref(TPtr data,
                     ex↲
tent_gen::gen_type<Num↲
Dims>::type ranges,
                       ↲
const storage_order& ↲
store = c_storage_or↲
der());
```

Effects. This constructs a `const_multi_array_ref` using the specified parameters. `ranges` specifies the shape and index bases of the constructed const_multi_array_ref. It is the result of `NumDims` chained calls to `extent_gen::operator[]`. `store` specifies the storage order or layout in memory of the array dimensions.

```
const_multi_ar↲
ray_ref(const ↲
const_multi_array_ref& ↲
x);
```

Effects. This constructs a shallow copy of `x`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Auxiliary Components

`multi_array_types`

```
namespace multi_array_types {
  typedef *unspecified* index;
  typedef *unspecified* size_type;
  typedef *unspecified* difference_type;
  typedef *unspecified* index_range;
  typedef *unspecified* extent_range;
  typedef *unspecified* index_gen;
  typedef *unspecified* extent_gen;
}
```

Namespace `multi_array_types` defines types associated with `multi_array`, `multi_array_ref`, and `const_multi_array_ref` that are not dependent upon template parameters. These types find common use with all Boost.Multiarray components. They are defined in a namespace from which they can be accessed conveniently. With the exception of `extent_gen` and `extent_range`, these types fulfill the roles of the same name required by MultiArray and are described in its concept definition. `extent_gen` and `extent_range` are described below.

`extent_range`

`extent_range` objects define half open intervals. They provide shape and index base information to `multi_array`, `multi_array_ref`, and `const_multi_array_ref` constructors. `extent_range`s are passed in aggregate to an array constructor (see `extent_gen` for more details).

Synopsis.

```
class extent_range {
public:
  typedef multi_array_types::index      index;
  typedef multi_array_types::size_type  size_type;

  // Structors
  extent_range(index start, index finish);
  extent_range(index finish);
  ~extent_range();

  // Queries
  index start();
  index finish();
  size_type size();
};
```

Model Of. DefaultConstructible,CopyConstructible

Methods and Types.

`extent_range(index start, index finish)`	This constructor defines the half open interval `[start,finish)`. The expression `finish` must be greater than `start`.
`extent_range(index finish)`	This constructor defines the half open interval `[0,finish)`. The value of `finish` must be positive.
`index start()`	This function returns the first index represented by the range

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`index finish()`	This function returns the upper boundary value of the half-open interval. Note that the range does not include this value.
`size_type size()`	This function returns the size of the specified range. It is equivalent to `finish()-start()`.

`extent_gen`

The `extent_gen` class defines an interface for aggregating array shape and indexing information to be passed to a `multi_array`, `multi_array_ref`, or `const_multi_array_ref` constructor. Its interface mimics the syntax used to declare built-in array types in C++. For example, while a 3-dimensional array of `int` values in C++ would be declared as:

```
int A[3][4][5],
```

a similar `multi_array` would be declared:

```
multi_array<int,3> A(extents[3][4][5]).
```

Synopsis.

```

template <std::size_t NumRanges>
class *implementation_defined* {
public:
  typedef multi_array_types::index index;
  typedef multi_array_types::size_type size_type;

  template <std::size_t NumRanges> class gen_type;

  gen_type<NumRanges+1>::type  operator[](const range& a_range) const;
  gen_type<NumRanges+1>::type  operator[](index idx) const;
};

typedef *implementation_defined*<0> extent_gen;
```

Methods and Types.

`template gen_type<Ranges>::type`	This type generator is used to specify the result of `Ranges` chained calls to `extent_gen::operator[].` The types `extent_gen` and `gen_type<0>::type` are the same.
`gen_type<NumRanges+1>::type operator[](const extent_range& a_range) const;`	This function returns a new object containing all previous `extent_range` objects in addition to `a_range. extent_range` objects are aggregated by chained calls to `operator[]`.
`gen_type<NumRanges+1>::type operator[](index idx) const;`	This function returns a new object containing all previous `extent_range` objects in addition to `extent_range(0,idx).` This function gives the array constructors a similar syntax to traditional C multidimensional array declaration.

Global Objects

For syntactic convenience, Boost.MultiArray defines two global objects as part of its interface. These objects play the role of object generators; expressions involving them create other objects of interest.

Under some circumstances, the two global objects may be considered excessive overhead. Their construction can be prevented by defining the preprocessor symbol `BOOST_MULTI_ARRAY_NO_GENERATORS` before including `boost/multi_array.hpp.`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`extents`

```
namespace boost {
  multi_array_base::extent_gen extents;
}
```

Boost.MultiArray's array classes use the `extents` global object to specify array shape during their construction. For example, a 3 by 3 by 3 `multi_array` is constructed as follows:

```
multi_array<int,3> A(extents[3][3][3]);
```

The same array could also be created by explicitly declaring an `extent_gen` object locally,, but the global object makes this declaration unnecessary.

`indices`

```
namespace boost {
  multi_array_base::index_gen  indices;
}
```

The MultiArray concept specifies an `index_gen` associated type that is used to create views. `indices` is a global object that serves the role of `index_gen` for all array components provided by this library and their associated subarrays and views.

For example, using the `indices` object, a view of an array `A` is constructed as follows:

```
A[indices[index_range(0,5)][2][index_range(2,4)]];
```

View and SubArray Generators

Boost.MultiArray provides traits classes, `subarray_gen`, `const_subarray_gen`, `array_view_gen`, and `const_array_view_gen`, for naming of array associated types within function templates. In general this is no more convenient to use than the nested type generators, but the library author found that some C++ compilers do not properly handle templates nested within function template parameter types. These generators constitute a workaround for this deficit. The following code snippet illustrates the correspondence between the `array_view_gen` traits class and the `array_view` type associated to an array:

```
template <typename Array>
void my_function() {
  typedef typename Array::template array_view<3>::type view1_t;
  typedef typename boost::array_view_gen<Array,3>::type view2_t;
  // ...
}
```

In the above example, `view1_t` and `view2_t` have the same type.

Memory Layout Specifiers

While a multidimensional array represents a hierarchy of containers of elements, at some point the elements must be laid out in memory. As a result, a single multidimensional array can be represented in memory more than one way.

For example, consider the two dimensional array shown below in matrix notation:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix}$$

Here is how the above array is expressed in C++:

```
int a[3][4] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
```

This is an example of row-major storage, where elements of each row are stored contiguously. While C++ transparently handles accessing elements of an array, you can also manage the array and its indexing manually. One way that this may be expressed in memory is as follows:

```
int a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
int s[] = { 4, 1 };
```

With the latter declaration of `a` and strides `s`, element `a(i,j)` of the array can be accessed using the expression

```
*a+i*s[0]+j*s[1]
```

.

The same two dimensional array could be laid out by column as follows:

```
int a[] = { 0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11 };
int s[] = { 3, 1 };
```

Notice that the strides here are different. As a result, The expression given above to access values will work with this pair of data and strides as well.

In addition to dimension order, it is also possible to store any dimension in descending order. For example, returning to the first example, the first dimension of the example array, the rows, could be stored in reverse, resulting in the following:

```
int data[] = { 8, 9, 10, 11, 4, 5, 6, 7, 0, 1, 2, 3 };
int *a = data + 8;
int s[] = { -4, 1 };
```

Note that in this example `a` must be explicitly set to the origin. In the previous examples, the first element stored in memory was the origin; here this is no longer the case.

Alternatively, the second dimension, or the columns, could be reversed and the rows stored in ascending order:

```
int data[] = { 3, 2, 1, 0,  7, 6, 5, 4, 11, 10, 9, 8 };
int *a = data + 3;
int s[] = { 4, -1 };
```

Finally, both dimensions could be stored in descending order:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
int data[] = {11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
int *a = data + 11;
int s[] = { -4, -1 };
```

All of the above arrays are equivalent. The expression given above for `a(i,j)` will yield the same value regardless of the memory layout. Boost.MultiArray arrays can be created with customized storage parameters as described above. Thus, existing data can be adapted (with `multi_array_ref` or `const_multi_array_ref`) as suited to the array abstraction. A common usage of this feature would be to wrap arrays that must interoperate with Fortran routines so they can be manipulated naturally at both the C++ and Fortran levels. The following sections describe the Boost.MultiArray components used to specify memory layout.

c_storage_order

```
class c_storage_order {
  c_storage_order();
};
```

`c_storage_order` is used to specify that an array should store its elements using the same layout as that used by primitive C++ multidimensional arrays, that is, from last dimension to first. This is the default storage order for the arrays provided by this library.

fortran_storage_order

```
class fortran_storage_order {
  fortran_storage_order();
};
```

`fortran_storage_order` is used to specify that an array should store its elements using the same memory layout as a Fortran multidimensional array would, that is, from first dimension to last.

general_storage_order

```
template <std::size_t NumDims>
class general_storage_order {

  template <typename OrderingIter, typename AscendingIter>
  general_storage_order(OrderingIter ordering, AscendingIter ascending);
};
```

`general_storage_order` allows the user to specify an arbitrary memory layout for the contents of an array. The constructed object is passed to the array constructor in order to specify storage order.

`OrderingIter` and `AscendingIter` must model the `InputIterator` concept. Both iterators must refer to a range of `NumDims` elements. `AscendingIter` points to objects convertible to `bool`. A value of `true` means that a dimension is stored in ascending order while `false` means that a dimension is stored in descending order. `OrderingIter` specifies the order in which dimensions are stored.

Range Checking

By default, the array access methods `operator()` and `operator[]` perform range checking. If a supplied index is out of the range defined for an array, an assertion will abort the program. To disable range checking (for performance reasons in production releases), define the `BOOST_DISABLE_ASSERTS` preprocessor macro prior to including multi_array.hpp in an application.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/