
MultiArray: A C++ library for generic programming with arrays



Ronald Garcia and Andrew Lumsdaine

Open Systems Lab, Indiana University, Bloomington, IN 47405, U.S.A.

SUMMARY

In C++, multi-dimensional arrays are often used but the language provides limited native support for them. The language, in its Standard Library, supplies sophisticated interfaces for manipulating sequential data, but relies on its bare-bones C heritage for arrays. The MultiArray library, a part of the Boost library collection, enhances a C++ programmer's tool set with versatile multi-dimensional array abstractions. It includes a general array class template and native array adaptors that support idiomatic array operations and interoperate with C++ Standard Library containers and algorithms. The arrays share a common interface, expressed as a generic programming *concept*, in terms of which generic array algorithms can be implemented. We present the library design, introduce a generic interface for array programming, demonstrate how the arrays integrate with the C++ Standard Library, and discuss the essential aspects of their implementation.

KEY WORDS: C++, array programming, generic programming, Standard Template Library, Boost

Introduction

Arrays are a ubiquitous feature of general-purpose programming languages. They play a prominent role in Fortran, the oldest language still in use today, and more modern programming languages like Java and C++ have them as well. Arrays in C++, like its other native data types, are inherited from C, a language with substantially less support for building abstractions. The C++ Standard Library [6, 15] (subsequently the Standard Library) builds more generally useful and usable components on top of legacy C data types. Containers like the `std::vector` and `std::deque` class templates hide inconvenient details of managing sequential data and export essential functionality for application developers, but the Standard Library supplies no similarly beneficial component for multi-dimensional arrays. The MultiArray library fulfills this role. It implements a general-purpose multi-dimensional array container and components for adapting native arrays to a similar interface. The library also lays a foundation for generic programming with arrays: it defines the MultiArray concept, a specification for array interfaces; generic algorithms implemented in terms of MultiArray apply to any array type from this library as well as any user-defined type that meets the concept requirements.

This paper provides an in-depth discussion of the MultiArray library with emphasis on its interfaces for array programming. We discuss some shortcomings of native C++ arrays and demonstrate how MultiArray addresses them. Generic programming philosophy, introduced briefly, guides an exposition of the MultiArray concept. Given an understanding of the fundamental array interface, we then study the interfaces of MultiArray's concrete array type, class template `multi_array`, and the adaptor classes for native C++ arrays, `multi_array_ref` and `const_multi_array_ref`. Their implementations require some generally useful advanced C++ techniques; we discuss these in detail.

Generic Programming

Generic programming is a paradigm for designing versatile, reusable, powerful libraries. It emphasizes the use of abstract interfaces to separate algorithms from the concrete data structures on which they operate. These interfaces are designed to match high-level characteristics of the problem domain and to allow for algorithm and data structure implementations that interoperate freely. Furthermore, the resulting algorithms are expressed in terms of domain-specific abstractions, rather than particular data types, easing program understanding and software maintenance. Since a generic algorithm does not manipulate concrete data structures directly, it can be applied to any particular data structure that conforms to its interface requirements.

The C++ Standard Library is considered a key example of generic programming in practice. In the Standard Library, the data structures are containers such as vectors and linked lists. *Iterators* form the abstract interface between algorithms and containers. Each Standard Library algorithm is written in terms of iterators and, as a result, can operate with any Standard Library container whose iterators meet the algorithm requirements. For instance, the `std::find` algorithm, implemented once, can be used to search `std::vector`, `std::list`, and `std::deque` containers with equal facility.

Generic programming terminology was popularized by the Standard Template Library [2, 10] and subsequently adopted for the C++ Standard Library and by C++ developers. The abstract interfaces that impose requirements on formal parameters to a generic algorithm are called *concepts* and any type that fulfills the requirements of a concept is said to *model* that concept. For example, pointer types such as `int*` model, or are models of, the Random Access Iterator concept as defined by the Standard Library; the class types `std::vector<double>` and `std::list<std::string>` model the Container concept.

Concepts define interfaces. They specify the *valid expressions* in which a model of the concept can appear. *Associated types* are the types of expressions involving concept models. Given a model of a concept, these type names can be referenced in terms of the model name. In C++ specifically, associated types are referenced either as nested type names within a concept model or using a *traits class* [11] to map from the concept model to a type name. Because generic programming emphasizes efficiency, *complexity guarantees* are often associated with the valid expressions. In this manner an algorithm's performance characteristics can be well specified in spite of its genericity. Then an algorithm's efficiency is not so much a function of its concrete arguments, but of the concepts used to implement it. Most concept specifications

contain further information that will not be of importance to what follows. (see [2]). To clarify when concepts are the topic of discussion, we typeset concept names in this paper using a sans serif font.

Consider the following implementation of the Standard Library `distance()` algorithm [4]:

```
template <class InputIterator>
typename std::iterator_traits<InputIterator>::distance_type
distance(InputIterator first, InputIterator last) {
    typename std::iterator_traits<InputIterator>::distance_type result = 0;
    for (; first != last; ++first)
        ++result;
    return result;
}
```

The Standard Library requires that the type of the arguments `first` and `last` models the Input Iterator concept. This concept has an associated type `distance_type`, which can be named using the `std::iterator_traits` traits class. The Input Iterator concept requirements ensure that `++first` and `first != last` are valid expressions. The algorithm imposes the additional requirement that `first` and `last` must delimit a valid range of values to ensure that the `for` loop terminates. The concept requirements alone do not guarantee this.

C++ does not provide language support for concepts. Template parameter names are not special: the template parameter name `InputIterator` is a placeholder for *any* type. C++ cannot enforce the concept requirements on the type used for `InputIterator`. Instead the function is instantiated using whatever type is passed and subsequently type checked. Naturally, if the type arguments do not support the expressions in the function body, an error will likely occur when type checking the body of that particular instantiation, but true concept checking would identify the choice of parameters as an error. That is, a C++ compiler will issue an error after instantiating a template whose body is invalid whereas true concept checking would issue an error without first instantiating the function. In practice, techniques like those found in [14] and [9] can be used to approximate concept checking for C++, but the language provides no native support for it.

Array Terminology and Notation

An array can be viewed as a nested hierarchy of containers. Its nesting depth determines its number of dimensions, or *dimensionality*. For instance, a one-dimensional array of integers is a container of integers; a two-dimensional array of integers is a container of one-dimensional arrays of integers; a three-dimensional array of integers is a container of two-dimensional arrays of integers; and so on. We call the arrays contained within another array its *subarrays*. A two-dimensional array has one-dimensional subarrays and similarly a three-dimensional array has two-dimensional and one-dimensional subarrays. The *immediate subarrays* of an array with dimensionality two or greater are the immediately contained subarrays. For instance, the immediate subarrays of a three-dimensional array are its two-dimensional subarrays alone.

We augment the language of the Standard Library containers with terminology that reflects the richer structure of arrays. To the standard notions of *values* and *value type*, we add

elements and **element type** The value type of an array or any other container is the type of its immediately contained values. The element type of an array is the type of the objects at its innermost nesting. To clarify, consider a three-dimensional array of integers. Its values are two-dimensional arrays of integers, but its elements are integers. For a one-dimensional array, the value type and element type are the same.

The **shape** of an array generalizes the Standard Library container notion of **size**. Size refers to the number of values in a container, but the shape of an array is the sequence of numbers denoting the size of the array and the subarrays at each nesting level. The shape sequence of an array begins with the number of immediate subarrays and proceeds inwards, ending with the size of its one-dimensional subarrays. The product of the shape values yields the total number of elements contained in an array. From this description, it follows that the size of an array is the same as the first component of its shape. For instance, a one-dimensional array containing five integers has shape 5 and size 5 (remember that the shape is a one integer sequence containing the number 5 whereas the size is the number 5; the notation is somewhat ambiguous). In contrast, a two-dimensional array that contains five one-dimensional arrays of three integers each has shape 5×3 but size 5.

The following table demonstrates how arrays are depicted throughout the paper:

Array	Contents	Shape	Dimensionality	Size																		
1	0 1 2	3	1	3																		
2	[[0 1 2] [3 4 5] [6 7 8]]	3×3	2	3																		
3	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>1</td><td>2</td></tr> <tr><td>3</td><td>4</td><td>5</td></tr> <tr><td>6</td><td>7</td><td>8</td></tr> </table>	0	1	2	3	4	5	6	7	8	3×3	2	3									
0	1	2																				
3	4	5																				
6	7	8																				
4	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>1</td><td>2</td><td>9</td><td>10</td><td>11</td></tr> <tr><td>3</td><td>4</td><td>5</td><td>12</td><td>13</td><td>14</td></tr> <tr><td>6</td><td>7</td><td>8</td><td>15</td><td>16</td><td>17</td></tr> </table>	0	1	2	9	10	11	3	4	5	12	13	14	6	7	8	15	16	17	$2 \times 3 \times 3$	3	2
0	1	2	9	10	11																	
3	4	5	12	13	14																	
6	7	8	15	16	17																	

Array #1 is one-dimensional and its values are delimited with brackets. Arrays #2 and #3 illustrate one array using two different layouts. The former uses brackets exclusively to delineate the array and its subarrays; the latter on the other hand uses a matrix notation: the entire array is bracketed and each row represents a subarray. Array #4 shows how the matrix notation can be generalized for higher-dimensional arrays, in this case a three-dimensional array. The outer brackets delimit the entire array while the inner brackets delimit the two 3×3 subarrays. We represent arrays using the generalized matrix notation.

Now suppose that Array #3 represents a matrix, a linear transformation. The rows of the matrix correspond to the subarrays of the array, but what of the columns? Some matrix operations require the independent manipulation of matrix columns, the subarrays cannot simultaneously represent the rows and columns. This calls for a means of addressing parts of an array along decompositions that differ from the subarray hierarchy. In our terminology, a **view** generalizes the notion of subarray to more arbitrary decompositions. Where a subarray specifically refers to a nested array within the array hierarchy, a view is any portion of an array, treated as though it were an independent array. In the general case, a view behaves like an array of references that point into the original array (bear in mind that arrays of references are not a legal C++ construct, though the idea is clear and instructive here). Accessing any

view element is equivalent to accessing the corresponding element in the original array. For example, the following is a 2×1 view of array #3:

$$\begin{bmatrix} 4 \\ 7 \end{bmatrix}$$

This example suggests several things about views. A view does not need to include all the elements along any dimension. Here the first dimension is limited to two values and the second dimension to one. A view's origin can be from anywhere in the original array. This example view begins in the middle of the array.

Index bases and addressing array values

Up to this point we have not talked about accessing an array's values. Each value of an array has associated with it an *index*. Likewise, each element of an N-dimensional array has an associated sequence of N indices. An index is an integral value that uniquely specifies a value in the array. If n is a valid index into an array, we use the notation $A[n]$ to refer to the value associated with that index. Indices are determined by the *index bases* of the array. Index bases are a list of integral values, one for each dimension, that specify the least valid index for a dimension. For instance, a 3×4 array with index bases $\{0, 0\}$ has as valid indices 0 through 2 for the first dimension and 0 through 3 for the second dimension. An array A with the above index bases addresses its first value as $A[0]$. The first value of this first value is $A[0][0]$, and so on. Because the elements of an array are found at the bottom of the nested array hierarchy, one can associate with each array element a sequence of N indices, where N is the array's dimensionality. For example, consider the following two dimensional array:

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Given the index bases $\{0, 0\}$, the array element 8 above can be referred to using the notation $A[2][2]$. Were the index bases $\{1, 1\}$, the same element would be addressed as $A[3][3]$. When discussion focuses on an element's indices, sequence notation is used. For example, if the above array has the former index bases, the element 8 is addressed using the indices $\{2, 2\}$.

Mapping array dimensions to memory

Though arrays conceptually have a rich hierarchical structure, they must somehow be mapped to a flat memory model. There are two common mappings to memory for multi-dimensional arrays. In one model, one-dimensional arrays are stored as contiguous blocks of memory; then, N-dimensional arrays are stored as contiguous blocks of pointers to (N-1)-dimensional arrays. This model requires a great deal of pointer chasing to access the elements of a higher-dimensional array. Another model, the focus of this section, stores all array elements in a single block of memory. Fortran and C++ both use this model to map native arrays to memory. The

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

Figure 1. Logical representation of a 4×4 array

order in which array elements are stored in memory depends on the programming language. Consider the two-dimensional array in Figure 1. When realized as a native C++ array, its elements map to memory as follows:

$$\begin{aligned} \hat{A} &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\} \\ &= \{A[0][0], A[0][1], A[0][2], A[0][3], A[1][0], A[1][1], A[1][2], \dots\} \end{aligned}$$

The above notation, used throughout this section, illustrates how logical arrays map to memory. The symbol \hat{A} signifies the memory mapping of array A . Linear sequences are depicted as comma-separated values enclosed in braces. The first line shows the ordering of the array elements in memory and the second line shows the indices of those elements in the same order. The example above shows that the array element ‘0’ is stored at some memory address and the remaining elements follow contiguously. This particular array layout is traditionally called “row-major” because the elements of each row are stored contiguously. The rows, then considered as units, are contiguous.

This mapping from array to linear memory corresponds to the following programmatic description:

```
int* elements = A_hat;
for(int i0 = 0; i0 != 3; ++i0)
  for(int i1 = 0; i1 != 4; ++i1)
    *elements++ = A[i0][i1];
```

This program describes how a block of memory **A_hat** would be filled with the array **A** elements. The variable **A_hat** represents the contiguous block of memory into which the elements of the array **A** are mapped. The **for** loops generate indices in the same order as depicted above, and the pointer **elements** is used to set the contents of **A_hat**. In general, mapping an N-dimensional C++ array to memory is equivalent to a loop nest where the first dimension is spanned by the outermost loop and each subsequent dimension is spanned by a loop nested within the loop for the previous dimension.

Fortran stores the same array in memory as follows:

$$\begin{aligned} \hat{A} &= \{0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11\} \\ &= \{A[0][0], A[1][0], A[2][0], A[0][1], A[1][1], A[2][1], \dots\} \end{aligned}$$

This arrangement is commonly called “column-major” because the elements of each column are stored contiguously. This mapping is equivalent to the following program:

```

int* elements = A_hat;
for(i1 = 0; i1 != 4; ++i1)
  for(i0 = 0; i0 != 3; ++i0)
    *elements++ = A[i0][i1];

```

Compared to the C++ array mapping, the order of the loops is reversed: the second dimension `i1` forms the outer loop and the first dimension `i0` forms the inner loop. Fortran mapping of N -dimensional arrays to memory is equivalent to a loop nest where the loop for each dimension is nested in the loop for the next dimension, the reverse of the C++ order. The last dimension forms the outermost loop.

The mapping from a logical array to linear memory is called the array's *storage order*, and it can be generalized beyond the two layouts described above. The storage order of an array is determined by two properties, *dimension order* and *dimension direction*. We use an example to explain these ideas. Suppose an N -dimensional array with shape $s_0 \times s_1 \times \dots \times s_N$ and index bases $\{b_0, b_1, \dots, b_N\}$. The mapping from logical array to memory will involve N nested loops, one for each dimension. Dimension order determines the order in which the dimensions are looped over. Specifically, dimension order is a list of numbers $\{d_0, d_1, \dots, d_N\}$ where each number specifies a dimension of the array. The value d_0 specifies the dimension for the innermost loop and d_N specifies the dimension for the outermost loop. To be more specific, if \mathbf{s} is an array of the N shape values, \mathbf{b} an array of the N index bases, \mathbf{i} an array of N index values, and \mathbf{d} an array representing the dimension order, then the C++ code that represents the mapping to memory takes the following form:

```

int* elements = A_hat;
for(i[d[N]] = b[d[N]]; i[d[N]] != b[d[N]] + s[d[N]]; ++i[d[N]])
  for(i[d[N-1]] = b[d[N-1]]; i[d[N-1]] != b[d[N-1]] + s[d[N-1]]; ++i[d[N-1]])
    ...
    for(i[d[0]] = b[d[0]]; i[d[0]] != b[d[0]] + s[d[0]]; ++i[d[0]])
      *elements++ = A[i[0]][i[1]]...[i[N]];

```

The array \mathbf{d} determines which dimension each loop spans. At each loop, the corresponding index variable $i[\mathbf{d}[\mathbf{K}]]$ is initialized to the first valid index for that dimension, $\mathbf{b}[\mathbf{d}[\mathbf{K}]]$, and then incremented each pass through the loop. The loop terminates when the index variable has taken on all valid index values, having looped $\mathbf{s}[\mathbf{d}[\mathbf{K}]}$ times. The dimension represented by $\mathbf{d}[0]$ will loop the fastest, and the dimension represented by $\mathbf{d}[N]$ will loop the slowest. More specifically, the value $i[\mathbf{d}[\mathbf{K}]]$ is incremented by one for every complete traversal of the valid values for $i[\mathbf{d}[\mathbf{K}-1]]$.

Dimension direction is a sequence of two possible values (here we use $+1$ and -1) that indicates if each dimension is stored in ascending index order ($+1$) or descending index order (-1). For example, map the array in Figure 1 to memory as follows:

$$\hat{A} = \{3, 2, 1, 0, 7, 6, 5, 4, 11, 10, 9, 8\}$$

$$= \{A[0][3], A[0][2], A[0][1], A[0][0], A[1][3], A[1][2], \dots\}$$

This corresponds to the following C++ code:

```

int* elements = A_hat;
for(i0 = 0; i0 != 3; ++i0)

```

```

for(i1 = 3; i1 != -1; --i1)
    *elements++ = A[i0][i1];

```

The dimension order of this layout matches row-major storage, but the mapping differs in that the second dimension is *reversed*, that is, stored in descending index order. Rather than looping from the least index to the greatest index, the loop proceeds from the greatest index to the least index. The dimension direction here is $\{+1, -1\}$. Now consider the array stored in row-major order with its first dimension reversed, that is, with dimension direction $\{-1, +1\}$:

$$\hat{A} = \{8, 9, 10, 11, 4, 5, 6, 7, 0, 1, 2, 3\}$$

$$= \{A[2][0], A[2][1], A[2][2], A[2][3], A[1][0], A[1][1], A[1][2], \dots\}$$

The corresponding code is as follows:

```

int* elements = A_hat;
for(i0 = 2; i0 != -1; --i0)
    for(i1 = 0; i1 != 4; ++i1)
        *elements++ = A[i0][i1];

```

Finally, we show both dimensions reversed (dimension direction $\{-1, -1\}$):

$$\hat{A} = \{11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0\}$$

$$= \{A[2][3], A[2][2], A[2][1], A[2][0], A[1][3], A[1][2], A[1][1], \dots\}$$

and the corresponding code:

```

int* elements = A_hat;
for(i0 = 2; i0 != -1; --i0)
    for(i1 = 3; i1 != -1; --i1)
        *elements++ = A[i0][i1];

```

A descending dimension changes the corresponding loop from:

```

for(i[d[K]] = b[d[K]]; i[d[K]] != b[d[K]] + s[d[K]]; ++i[d[K]])

```

to

```

for(i[d[K]] = b[d[K]]-1 + s[d[K]]; i[d[K]] != b[d[K]]-1; --i[d[K]])

```

thereby looping from the greatest valid index to the least valid index.

Shortcomings of native arrays

Native C++ arrays exhibit certain limitations. Several issues, discussed below, arise when native arrays are considered in the greater context of C++ software construction.

Standard Library compatibility

One fundamental feature of the Standard Library is its use of iterator concepts to interface data structures and algorithms. Most Standard Library algorithms use iterators to address a range of values, and all Standard Library containers supply `begin()` and `end()` member functions for creating iterators that bind them to the algorithms.

C++ native pointers model the Random Access Iterator concept and as such can be used with many Standard Library algorithms, but the resulting code can be fragile and error prone. Consider the following code to apply a function, `array_op()`, to the three 3×2 subarrays of a $3 \times 3 \times 2$ array:

```
double A[3][3][2];
// ...
std::for_each(&A[0],&A[3],array_op);
```

The `for_each()` call takes two pointers, which delimit the beginning and (one past) the end of the immediate subarrays, and a pointer to the function `array_op()`. The algorithm calls `array_op()` three times, once for each immediate subarray. The first two arguments to `for_each()` are unstable: when the shape of `A` changes, the function arguments must change accordingly. In contrast, if the array is replaced with a Standard Library container, then the equivalent call:

```
// A may be an std::vector, std::deque, std::list, etc.
std::for_each(A.begin(),A.end(),container_op);
```

makes no reference to the container's size. The `begin()` and `end()` member functions provide iterators while hiding implementation details.

C++ native arrays and Standard Library containers combine only in limited ways. One can declare native arrays of containers, and they work as expected:

```
std::vector<int> X[3]; // Ok
X[1].push_back(5); // Ok
```

but containers of native arrays have certain restrictions, as shown below:

```
typedef int array3[3][3][3]; // three-dimensional array type
array3 datum; // array instance

std::vector<array3> A; // Ok
A.push_back(datum); // Compiler Error!
std::vector<array3> B(3,datum); // Ok
std::vector<array3> C(3); // Compiler Error! Need explicit initializer.
```

The array `A` illustrates that a vector of arrays can be declared and default constructed but additional arrays cannot be inserted into the vector. Alternatively, as shown by array `B`, a vector can be constructed with a specified size, but the constructor requires an explicit initializer.

Crafting generic array algorithms

The `for_each` example above treats a three-dimensional array as a three-value sequence of two-dimensional arrays. In general, C++ native arrays work with generic algorithms that expect sequences, but generic array algorithms require information about the entire array, including its dimensionality, shape, and total number of elements. Generic algorithms can be written in terms of native arrays, but their versatility is undermined by limitations in the native array interface. One can implement an array algorithm using a completely generic interface, such as

```
template <class Array> void array_algorithm(Array&);
```

where it is assumed that `Array` denotes a native array type, but the shape and dimensionality of the array is unavailable to the function implementation. Another technique for writing array algorithms specializes the algorithm for each dimensionality, as in:

```
template <class T, int N1> void array_algorithm(T(&)[N1]);
template <class T, int N1, int N2> void array_algorithm(T(&)[N1][N2]);
```

and so on. This method retains the shape and the element type of the array at the cost of code duplication and the lack of dimension-independent genericity.

Working with array views

Native C++ arrays do not provide any convenient mechanisms for manipulating array views. Consider a three-dimensional array with shape $3 \times 3 \times 2$:

$$\left[\begin{array}{cc} [0 & 1] \\ [2 & 3] \\ [4 & 5] \end{array} \quad \begin{array}{cc} [6 & 7] \\ [8 & 9] \\ [10 & 11] \end{array} \quad \begin{array}{cc} [12 & 13] \\ [14 & 15] \\ [16 & 17] \end{array} \right]$$

and its representation as a native array:

```
int A[3][3][2] =
    { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17};
```

This native array can be treated as a sequence of three 3×2 subarrays. For example, `A[1]` refers to the subarray:

$$\begin{array}{cc} [6 & 7] \\ [8 & 9] \\ [10 & 11] \end{array}$$

However, the array cannot easily be viewed along a different decomposition. For instance, consider the first column of each subarray, a 3×3 array in its own right:

$$\begin{array}{ccc} [0 & 6 & 12] \\ [2 & 8 & 14] \\ [4 & 10 & 16] \end{array}$$

C++ native arrays provide no means to regard this view as a first-class array.

Language interoperability

C++ native arrays lack two features that leverage knowledge and capabilities offered by components developed for other programming languages. First, C++ native arrays have fixed index bases. They use 0-based indexing, meaning that the index bases are always zero. However, 0-based indexing is not always ideal. In Fortran, for instance, arrays are 1-based; thus, when translating Fortran programs to C++ or when implementing array algorithms described using 1-based indexing, as they often appear in literature, it may be preferable to implement using this other indexing scheme.

Second, when writing programs where arrays interact across language boundaries, arrays must be stored in a compatible manner. If an array created by a C++ program is passed to a Fortran subroutine, the array data is interpreted differently by the two languages. If proper care is not exercised in this situation, the result is likely to be undesirable. Many numerical linear algebra libraries, such as LAPACK [1] and the Basic Linear Algebra Subprograms [8] are written in Fortran and represent linear algebra structures, like vectors and matrices, using arrays. These libraries have matured and stabilized over many years of use and maintenance. Rather than reinvent the wheel, many C++ programs directly call routines from these libraries for scientific computing. One difficulty with doing this is that, as previously discussed, Fortran arrays are stored in memory differently than C++ native arrays. Usually the C++ program explicitly accounts for reversed dimensions to facilitate calls to Fortran routines. This muddles the implementation because the array layout in memory complicates the C++ program, which no longer reflects the logical array in the source code. C++ native arrays have no means to control the storage order independent of the logical array interface.

The MultiArray concept

Generic programming focuses on the specification of abstract domain-specific interfaces. In light of this, it follows naturally that the MultiArray concept is the cornerstone of this library. Since all the arrays in this library model the concept, its valid expressions and associated types can be used to implement generic algorithms that work with any MultiArray component. In this section, we introduce the expressions and types that comprise the generic array interface specified by the MultiArray concept.

Throughout the discussion that follows, the type name **Array** represents a model of the MultiArray concept. It is a placeholder for any valid model of the concept, to which the discussion applies. For brevity, member function declarations and type names omit the **Array::** qualifier. Unless otherwise indicated, all non-native types and functions required by MultiArray are declared as nested types and member functions.

Accessing values and manipulating subarrays

Native C++ arrays use brackets to address subarrays and MultiArray supports the same notation. Models of MultiArray must implement the following member functions:

```
reference operator[](index idx);
```

```
const_reference operator[](index idx);
```

The `index` type is a signed integral type used to index arrays. The associated type names `reference` and `const_reference` suggest, correctly, that the values returned from bracket operators refer to the parent array. An object of `reference` type is mutable and modifications to it will affect the array to which it refers. The `const_reference` type is not mutable and as such can only be used to inspect the contents of the referent array. Another type worth mentioning here is the `value_type` associated type. The result of calling `operator[]()` can also be assigned to a `value_type` variable. Unlike `reference` objects, Modifications to `value_type` variables do not affect the original array. The following code demonstrates these distinctions:

```
template <class Array>
void modify_value(Array& A) {
    typename Array::reference ref = A[0];
    typename Array::const_reference cref = A[0];
    typename Array::value val = A[0];
    modify(ref); // Will cause changes to A
    modify(val); // Will NOT change A
    modify(cref); // Error! Can't modify cref.
}
```

The `modify()` function alters the contents of its array argument. Passing `ref` to the function results in changes to the array `A`, whereas passing `val` does not because it is an independent array. Passing `cref` results in a compile error because mutation through `const_reference` types is prohibited.

In many applications, an N-dimensional array's elements are accessed directly in terms of the array. The following code illustrates setting the elements of a three-dimensional array:

```
template <class Array>
void set_array(Array& A) {
    // A is a three-dimensional array
    for(int i = 0; i != 4; ++i)
        for(int j = 0; j != 4; ++j)
            for(int k = 0; k != 4; ++k)
                A[i][j][k] = 5.0;
}
```

This interface matches that of native C++ arrays for accessing elements.

The `MultiArray` concept explicitly supports subarrays and provides facilities for creating and manipulating them. The concept defines nested type generator `subarray` and `const_subarray`, which name subarrays of any dimensionality. A specific relationship exists between these subarray types and the `reference` type of an array. For an N-dimensional array, the `reference` type and the type `subarray<N-1>::type` are synonymous; similarly, the `const_reference` and `const_subarray<N-1>::type` types are also the same. The type generator is useful when the dimensionality of an array and the desired subarray is known at compile time as in the following:

```
template<class Array>
void clear_intarray3(Array& A) {
    // A is known to be a 3-dimensional array of ints
    typename Array::iterator i;
```

```

// typename Array::reference::iterator j
typename Array::typename subarray<2>::type::iterator j;
// typename std::iterator_traits<Array::reference::iterator>::reference::iterator k
typename Array::typename subarray<1>::type::iterator k;
for(i = A.begin(); i != A.end(); ++i)
  for(j = i->begin(); j != i->end(); ++j)
    for(k = j->begin(); k != j->end(); ++k)
      *k = 0;
}

```

In the above example, the `subarray` type generator is not always the most concise means of naming a subarray type, but the type name remains constant in length relative to the nesting depth and dimensionality of the subarray type. This can be important for generic programs that determine dimensionality based on compile-time constants.

To clarify, the following table shows some examples of subarrays and other types that result from value access expressions:

Expression	Type	Result									
<code>A</code>	<code>Array</code>	<table border="1"> <tr><td>0</td><td>1</td><td>2</td></tr> <tr><td>3</td><td>4</td><td>5</td></tr> <tr><td>6</td><td>7</td><td>8</td></tr> </table>	0	1	2	3	4	5	6	7	8
0	1	2									
3	4	5									
6	7	8									
<code>A[0]</code>	<code>Array::subarray<1>::type</code>	<table border="1"> <tr><td>0</td><td>1</td><td>2</td></tr> </table>	0	1	2						
0	1	2									
<code>A[1]</code>	<code>Array::subarray<1>::type</code>	<table border="1"> <tr><td>3</td><td>4</td><td>5</td></tr> </table>	3	4	5						
3	4	5									
<code>A[0][1]</code>	<code>Array::element</code>	1									

Fetching Iterators

The MultiArray concept requires the following expressions and associated types for creating iterators:

```

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

```

The `begin()` and `end()` member functions return iterators pointing to the first and (one past) the last value of an array respectively. The `rbegin()` and `rend()` member functions also return iterators that delimit the array values, but they differ in that the order of traversal is reversed: the iterator that `rbegin()` returns points to the last value of the array and the `rend()` member function returns the corresponding iterator pointing to (one before) the first value. Observe that array iterators access array values, which only coincide with elements for one-dimensional arrays. For example, though the elements of a three-dimensional array of integers are integers, the `begin()` member function, when called on this array, returns an iterator over two-dimensional subarrays of integers.

MultiArray iterators reinforce the modelling of arrays as nested hierarchies of containers. Consider the following program that assigns to the elements of a two-dimensional array.

```
template <class Array>
void assign_Array (Array& A) {
    // A is a two-dimensional array of doubles.
    typedef typename Array::iterator iterator2;
    typedef typename iterator2::value_type::iterator iterator1;
    for(iterator2 i = A.begin(); i != A.end(); ++i)
        for(iterator1 j = i->begin(); j != i->end(); ++j)
            *j = 5.0;
}
```

This program will work equally well for a model of MultiArray and for an object of type `std::vector< std::vector<double> >`. The latter type does not model MultiArray, but its iterators are equivalent. Clearly the MultiArray iterators and Standard Library container iterators behave similarly.

Comparing arrays

The `std::vector` class template implements two comparison operators, `operator==()` and `operator<()`. Two vectors are considered equal if they contain the same number of values and if corresponding values between the vectors are equal. The less-than relationship between vectors is defined *lexicographically*. Operationally, this means that contained values are compared between two vectors, from the first to the last. On the first occasion that two values are not equal, the vector with the lesser value is judged to be less than the other vector. Both equality and comparison operators on vectors depend on those same operators being defined for the container's value type. In the Standard Library, this is expressed by requiring that the type models the Less Than Comparable and Equality Comparable concepts.

Models of the MultiArray concept support the above operators and the following logical extensions of them as well:

```
bool operator<(Array& rhs);
bool operator==(Array& rhs);
bool operator<=(Array& rhs);
bool operator>(Array& rhs);
bool operator>=(Array& rhs);
bool operator!=(Array& rhs);
```

In line with the behavior of the Standard vector class, these operators treat arrays as lexicographically ordered. In other words, arrays are ordered lexicographically based on the ordering of the contained values. Since arrays of dimensionality 2 or higher have subarrays for values and subarrays define their comparison operators lexicographically as well, comparison is defined as a recursion of lexicographical comparisons that proceed down the nested array hierarchy. In other words, comparison of two arrays results in a lexicographical comparison of their immediate subarrays, each of which is in turn compared lexicographically, and so forth.

The following table illustrates equality and less-than relations between several arrays.

Example	Comparison
1	<code>[5 5 5] == [5 5 5]</code>
2	<code>[5 5 5] != [5 5 3]</code>
3	<code>4 < 5</code>
4	<code>[5 5 4] < [5 5 5]</code>
5	<code>[1 2 3; 5 5 4; 4 5 6] < [1 2 3; 5 5 5; 4 5 6]</code>

The first and second examples demonstrate a general property: two arrays are equal if all their corresponding elements are equal; conversely, if any two corresponding elements differ, the arrays are not equal. The third example compares one-dimensional arrays of shape 1. The comparison in this case is analogous to the comparison of the contained single elements. The fourth example compares two one-dimensional arrays lexicographically. Though the first two values of the arrays – which happen to also be elements – are equal, the third values are not. Since the third value of the left hand array is less than the third value of the right hand array, the left hand array is judged the lesser. The final example compares two-dimensional arrays. In this case the array values are the row subarrays. To compare the two arrays, the respective first values, specifically the first rows, are compared and found to be equal. Then the second values, the second rows, are compared. The last value of the left hand array's second row is less than the corresponding value in the right hand array. As a result, the left hand array is deemed the lesser of the two. This example illustrates how nested array comparisons order higher-dimensional arrays. Bear in mind that array comparisons depend on the definition of the comparison operators for the element type. The `MultiArray` concept only defines comparison expressions for arrays when the element type models both the `Equality Comparable` and `Less Than Comparable` concepts.

Creating Views

The `MultiArray` interface for creating views is similar to that for creating subarrays. The concept defines nested view type generators `array_view` and `const_array_view` and overloads `operator[]()` to return views when passed a list of ranges built using an object of type `index_gen`. The `index_gen` type overloads its own `operator[]()` to take *index range* values, objects of nested type `index_range`. When creating a view of an array, the index range objects specify for each dimension the indices to include in the resulting view. The index range constructor is of the form:

```
index_range(index start, index finish, index stride=1)
```

The **start** value specifies the first index value from the original array to include from the corresponding dimension. The **finish** value is the index one past the last value to include. Finally, the stride specifies the distance between included values. For example, a stride of 1 includes every element within the range; a stride of 2, on the other hand, includes every other element in the range. Index ranges specify half-open ranges and the number of indices included

in the resulting view for the corresponding dimension is $(\text{finish}-\text{start})/\text{stride}$. When an **index_range** of the resulting view specifies only one valid index, the view can be specified to have one dimension. If the **index_range** object is replaced with the integer representing the valid index, the resulting view will have one less dimension.

The following code sequence illustrates the creation of a view from an existing array.

```
template <class Array>
typename Array::array_view<2>::type
make_view (Array& A) {
    typedef typename Array::index_range range;
    typename Array::index_gen indices;

    // if
    // A == [[0 1 2] [3 4 5] [6 7 8]]
    // then
    // A_view0 == [[0 1] [3 4]]
    typename Array::array_view<2>::type A_view0 =
        A[indices[range(0,2)][range(0,2)]];
    return A_view0;
}
```

Here, the **index_range** type is aliased for more convenient use. The **indices** object is used to specify two ranges of indices, one for each dimension of the original array, and these ranges are passed to the **operator[]()** of the array. The result is a two-dimensional view, named using the **array_view** type generator. To clarify, the following table shows some more example views of the same array:

Expression	Type	Resulting Array
A	Array	0 1 2 3 4 5 6 7 8
A[indices[range(1,3)][range(0,2)]];	Array::array_view<2>::type	3 4 6 7
A[indices[range(0,3,2)][range(0,3,2)]];	Array::array_view<2>::type	0 2 6 8
A[indices[range(0,3)][range(0,1)]]	Array::array_view<2>::type	0 3 6
A[indices[range(0,1)][range(0,3)]]	Array::array_view<2>::type	0 1 2
A[indices[range(0,3)][0]]	Array::array_view<1>::type	0 3 6
A[indices[0][range(0,3)]]	Array::array_view<1>::type	0 1 2

The first example focuses on a 2×2 corner of the array. The second example shows how strides affect view construction. In this manner, one can choose to view every n th element along a certain dimension. In this example specifically, the view covers only the corner elements of the array by viewing every other element along both dimensions. The third example covers all indices along the first dimension and only the first valid index, 0, for the second dimension. The resulting array still has 2 dimensions, though the second is redundant because 0 is the only

valid index. The fourth example reverses the range specifications from the third example. The result, though visually different by our notation, is not interesting otherwise. Here, though, the first dimension is fixed to an index of 1, while the second dimension can range over the same indices as the original, 0 to 2. The last two examples illustrate how an integral value can be used to create a view of fewer dimensions. Each specifies one index to be a fixed value. In both cases, the result is a one-dimensional array and the fixed dimension no longer applies.

These views have certain limitations compared to the more general description of views given earlier. This interface only supports views that can be specified using ranges and uniform striding. Techniques exist to create more generalized views. Indirect indexing, for instance, permits the creation of more arbitrary views of an array at the expense of performance (an extra pointer dereference for each element access) and space (an indirect array requires space proportional to the number of elements it references). The design used here covers most interesting cases while minimizing complexity and cost.

Array Representation

The MultiArray concept places requirements on an array's physical representation in memory. An array's memory layout is determined by three attributes: strides, index bases, and origin.

Multi-dimensional arrays have a nested structure, and as such they must be mapped to linear memory. This mapping is represented by an array's *strides*. The strides are a sequence of integers that describe how an array's elements are distributed in memory. Each stride associates with a dimension of an array an integral factor that determines the distance between adjacent elements along that dimension. In other words, given an array element with specific indices, $\{i_1, \dots, i_n, \dots, i_N\}$, the n th stride indicates how many positions forward (or backward) to move in memory to find the element of the array with indices $\{i_1, \dots, i_n + 1, \dots, i_N\}$ (or $\{i_1, \dots, i_n - 1, \dots, i_N\}$).

To better understand strides, consider a two dimensional array:

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

and a linear mapping of it to memory:

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$$

By convention, the first dimension of the array selects a row from the illustration and the second dimension selects a column, or an element of a row. The stride sequence for this array layout is $\{3, 1\}$. To move forward d positions along the n th dimension requires a pointer offset of $d * n$. For instance, to move 1 place forward along the first dimension requires a pointer offset of $3 * 1 = 3$. Indeed the array elements '4' and '7' are 3 positions apart in the linear layout, but '4' and '5' are only $1 * 1 = 1$ position apart. The array's strides in this case declare that to move along the first dimension ("vertically") in the array requires a factor of 3 change in the pointer offset, but movement along the second dimension ("horizontal") only requires a factor of 1.

An array's *index bases* specify the starting index for each dimension. Though C++ native arrays are always 0-based, meaning the indices all start at 0, it is sometimes more convenient to use a different indexing scheme. For example, Fortran arrays are always 1-based; thus, when translating Fortran code to C++, it is often easier to use Fortran style 1-based indexing to implement the C++ code. Each index basis indicates, for its dimension, the first valid index for that dimension. In the case of a Fortran-style array, all index bases are 1.

The *origin* of an array is the address of the array element whose index is zero for all dimensions, as in $\mathbf{A}[0]$ for one dimensional arrays, $\mathbf{A}[0][0]$ for two dimensional arrays, and so on. Alternatively, if the index bases are less than zero, the origin of the array will be somewhere in the "middle" of the elements, that is, neither the first nor last element in the linear element list.

The strides, shapes, and index bases together uniquely determine the location of an array's elements in memory. Given an N -dimensional array with shape sequence $shape_i$, starting data address $addr$, strides $stride_i$, and index bases $base_i$, the address of the element associated with the indices $index_i$, can be calculated using the formula

$$element_address = origin + \sum_{i=1}^N index_i * stride_i$$

where

$$origin = addr + \sum_{i=1}^N -base_i * stride_i$$

and

$$base_i \leq index_i < base_i + shape_i.$$

To clarify, consider again the array described above and suppose the following values for the array's representation:

$$\begin{aligned} shape &= \{3, 3\} \\ stride &= \{3, 1\} \\ base &= \{0, 0\} \end{aligned}$$

and assume that the top left element of the array (The "0" element) is the origin (*origin*). Then the element $\mathbf{A}[1][2]$ (the '5' element), meaning

$$index = \{1, 2\},$$

is located at address:

$$origin + (1 - 0) * 3 + (2 - 0) * 1 = origin + 5$$

MultiArray specifies the following interface for querying an array's internal representation and shape:

```

Array::element* origin();
Array::element const* origin() const;
Array::size_type const* shape();
Array::index const* strides();
Array::index const* index_bases();

```

The `size_type` type represents the unsigned integral values used to describe the shape of an array. The `shape()`, `strides()`, and `index_bases()` member functions each return a pointer to the beginning of a native C++ array that holds the associated sequence values; the `origin()` member function returns the address of the array's origin.

The following function template implements the formula described above for accessing elements. It takes an array and a native array of indices and returns a reference to the associated array element.

```

template <class Array>
typename Array::element& access(Array& A, typename Array::index* idxes) {
    typename Array::element* address = A.origin();
    typename Array::index const* strides = A.strides();
    for(int i = 0; i != A.num_dimensions(); ++i)
        address += idxes[i]*strides[i];
    return *address;
}

```

An example generic algorithm

To bring together our discussion of the MultiArray concept, we present the following generic array algorithm:

```

template <typename Element>
struct print_array_dispatch {
    template <typename Array>
    static std::ostream&
    go(std::ostream& os, const Array& A) {
        typename Array::const_iterator i;
        os << "[";
        for (i = A.begin(); i != A.end(); ++i) {
            go(os, *i);
            if (i+1 != A.end())
                os << ',';
        }
        return os << "]";
    }

    static std::ostream&
    go(std::ostream& os, const Element& x) {
        return os << x;
    }
};

template <typename Array>
std::ostream& print_array(std::ostream& os, const Array& A) {

```

```

    return print_array_dispatch<typename Array::element>::go(os,A);
}

```

It implements a generic output routine for streaming arrays. The `print_array()` function defers to the `print_array_dispatch` class template for its operations. The class template overloads the `go()` member function for all array types and for the array elements. Because every MultiArray array type models the MultiArray concept, the `print_array()` algorithm function works for all of them. Additionally, `print_array()` demonstrates an idiom for developing generic array algorithms that recur over subarray structure.

The `multi_array` class template

The `multi_array` class template implements a general-purpose multi-dimensional array class. This section describes the features of `multi_array` including constructors, access operations, copy and comparison operations, and mutation operations such as resizing and reshaping. Some `multi_array` operations are shared by all MultiArray library arrays, in which case they are discussed under the MultiArray concept. The `multi_array` class is defined in the Boost library header file `boost/multi_array.hpp`, which supplies all the needed components to use `multi_array`. Additionally, the concrete MultiArray components are encapsulated within the `boost` namespace. For the purpose of brevity, we omit the namespace prefix on components. It is thus assumed that the namespace was imported into the global namespace using the directive `using namespace boost;`

Template parameters

The `multi_array` class template is declared as follows:

```

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

```

The `Element` type parameter denotes the *element type* of an array, the specific type of the objects that are stored in the array. The `NumDims` parameter to `multi_array` determines the type's dimensionality. Though the shape of an array is not part of an array's type, its dimensionality is. For example, a 3×3 array has dimensionality 2 and as such may have the same array type as a 5×5 array.

The `Allocator` type parameter refers to the class that `multi_array` uses for memory management. By default this type is `std::allocator`, which is the desired type in most cases. Under certain circumstances, however, the library user may wish to customize memory behavior by supplying a user-defined allocator class.

Constructors

A `multi_array` object can be constructed in several ways. The constructor declarations follow:

```

multi_array();

```

```

template <typename ExtentList>
explicit multi_array(const ExtentList& sizes,
                    const storage_order& store = c_storage_order(),
                    const Allocator& alloc = Allocator());

explicit multi_array(const extents_gen::gen_type<NumDims>& ranges,
                    const storage_order& store = c_storage_order(),
                    const Allocator& alloc = Allocator());

multi_array(const multi_array& x);

```

The default constructor creates an empty array, that is, an array with shape $0 \times 0 \times \dots \times 0$. This is useful when the desired shape of an array is unknown at creation time. Later we discuss a mechanism for resizing an existing array.

Another constructor specifies the shape of the array. For example, a $4 \times 4 \times 4$ array of **doubles** is constructed using the following expression:

```
multi_array<double,3> A(extents[4][4][4]);
```

The syntax is comparable to the syntax for creating a native array:

```
double A[4][4][4];
```

The **extents** singleton [5] object overloads **operator[]()** to accumulate shape information for an array.

The **multi_array** class has a constructor that accepts any type that models the **Collection** concept [13]. This concept describes classes that supply **begin()** and **end()** member functions for accessing iterators and nested type definitions for contained value and reference types. **Collection** is less strict a concept than the **Sequence** concept from the Standard Library. The following example uses an **std::vector** (which models **Collection**) to construct a $4 \times 4 \times 4$ array:

```
std::vector<int> idx(3,4);
multi_array<double,3> A(idx);
```

Observe that the first expression above constructs a three-object vector that contains 3 copies of the number 4. This example uses a **std::vector** of integers to construct the array.

The constructors described above take additional parameters, but the defaults usually suffice. A **multi_array** object, like other container classes in the Standard Library, can be constructed with an allocator object. This parameter, the third, defaults to an allocator constructed with no parameters.

The second parameter to an array is a **storage order** object, which determines how an **multi_array**'s elements are laid out in memory. The **multi_array** class supports three possible values for the layout parameter. The default layout option, **c_storage_order** lays out the array using the same layout as a native C++ array, that is, in “row-major” order. The **fortran_storage_order** option lays out an array in the same form as the Fortran language lays out arrays, “column-major” order. A custom storage order can be supplied using the **general_storage_order** class. This class is constructed with two parameters. The first is a sequence of numbers that dictate the dimension order in which the array is to be stored. The second parameter is a sequence of booleans that determine for each dimension the direction in which the values are stored. A **true** value indicates ascending direction and a **false** value

indicates a descending direction. Both parameters must be models of the Collection concept. The following example uses two `std::vector` objects to create a `general_storage_order` object and construct an array that uses that storage order:

```
int const order_size = 3;
int order_data[order_size] = {0, 1, 2};

bool const direction_size = 3;
bool direction_data[direction_size] = {true,true,true};

std::vector<int> order(order_data,order_data+order_size);
std::vector<bool> direction(direction_data,direction_data+direction_size);

multi_array<double,3> A(extents[3][3][3],
                      general_storage_order<3>(order,direction));
```

The array `A` is constructed using the `general_storage_order` parameter to dictate the storage order. In this case, the array has the same storage order as would result from using the `fortran_storage_order`. With respect to the `MultiArray` concept, the shape and storage order of an array combine to determine its strides.

Changing index bases

By default, a `multi_array` is 0-based, but the library offers facilities for altering index bases. Index bases can be set to nonzero values during construction. In addition to integral values, the `extents` object can also be given objects of type `extent_range`. These objects are constructed to specify a half-open interval of valid indices using the constructor `extent_range(start,finish)`. The value `start` indicates the least valid index, and the value `finish` is one past the greatest valid index. Thus `finish-start` elements will exist in this range. Passing an integer `k` to `extents` is a shorthand for `extent_range(0,k)`. Consider the following code:

```
typedef multi_array<double,3>::extent_range range;
multi_array<double,3> A(extents[range(1,4)][range(1,4)][range(1,4)]);
```

It creates an array where each index basis is 1, and the array has shape $3 \times 3 \times 3$.

The `multi_array` class also supports adjusting the index bases after construction using the `reindex()` member function. When passed an integral value, `reindex()` sets all index bases to that value, as shown below:

```
multi_array<double,3> A(extents[3][3][3]);
A.reindex(1);
```

This example sets all the index bases to 1. The `reindex()` function is also overloaded to accept any model of the Collection concept. This is used to specify a distinct index base for each dimension. The following code demonstrates this use:

```
multi_array<double,3> A(extents[3][3][3]);
int A_bases_data[] = {-1, 0, 1};
std::vector<int> A_bases(&A_bases_data[0],&A_bases_data[3]);
A.reindex(A_bases);
```

The array **A** ends up with different index bases for its three dimensions, -1, 0, and 1 respectively.

Assignment

Once a **multi_array** object is constructed, it may be assigned elements in several ways. To assign a single element at a time, **operator[]()** is generally used, as in the following:

```
A[0][0] = 7;
```

The class additionally supports a mechanism for setting the entire array using a sequence of elements. The **assign()** member function takes two iterators as parameters and copies the represented representation of elements into the array. For example, a 3×3 array of integers can be initialized to known values using the following sequence of instructions:

```
int data[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8 };
int size = sizeof(data)/sizeof(int);
multi_array<int,2> A(extents[3][3]);
A.assign(data,data + size);
multi_array<int,2> B(extents[3][3],fortran_storage_order());
B.assign(data,data + size);
assert(A != B);
```

The iterators passed to **assign()** must traverse the same number of elements as are contained in the array. In addition, bear in mind that the elements are copied verbatim into the memory block of the array. For this reason, arrays with different storage orders differ if assigned the same block of elements. In the above example, **B** does not have the same value as **A**.

Copying values

The **multi_array** class has deep assignment and copy semantics. If an array is passed to a function by value, a new array with copies of all contained elements is constructed. Arrays passed by reference to functions minimize copy overhead. A **multi_array** can be assigned from any other **multi_array** with the same shape. If two arrays have the same dimensionality but different shape, assignment is undefined.

```
multi_array<double,1> A(extents[3]), B(extents[3]);
// ...
// A has been assigned values
B = A;
assert(A == B);
```

Data

Some applications require direct access to the contents of a **multi_array**. In order to pass a **multi_array** to a Fortran subroutine, for example, a pointer to the array's contents is needed. The elements of a **multi_array** are stored in a contiguous portion of memory. A pointer to that data may be retrieved from an array using the **data()** member function, which returns a

pointer to the first element of the array. Some array operations move a **multi_array**'s storage, invalidating any previously recorded pointers to data. The **data()** member function is not always the same as the **origin()** member function, though the two coincide for any array with zero-valued index bases and ascending dimension direction. Nonzero index bases or descending dimensions can lead to differences between the two functions.

Resizing arrays

By default a **multi_array** retains its shape for the duration of its lifetime, but a user may require this to change. One means for changing the shape of an array is to resize it. The **resize()** member function accepts the same shape parameter as the array constructor. The array is altered to have the new shape, which must have the same dimensionality as the previous shape. The array also preserves elements that overlap the previous shape description. If each dimension is given a strictly nondecreasing shape, then all the original elements are preserved and new elements are default constructed to fill the new space. If some dimensions shrink in extent, then some elements will be lost.

To demonstrate, the following table illustrates several resize expressions:

Array A Before	Resize Expression	Array A After
$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$	A.resize(extents[3][3]);	$\begin{bmatrix} 0 & 1 & 0 \\ 2 & 3 & 0 \\ 0 & 0 & 0 \end{bmatrix}$
$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$	A.resize(extents[2][1]);	$\begin{bmatrix} 0 \\ 2 \end{bmatrix}$
$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$	A.resize(extents[1][2]);	$\begin{bmatrix} 0 & 1 \end{bmatrix}$

The first expression enlarges a 2×2 array to shape 3×3 , filling in the new space with default-constructed values (which are 0 in this specific example). The second expression shrinks the array down to a 2×1 array, constricting the second dimension. Finally, the third expression shrinks the first dimension of the array down so that the array shape is 1×2 . In the case of the last two expressions, some of the elements of the original array are lost.

Reshaping arrays

Resizing an array is not the only means by which its shape can change. The **reshape()** member function remaps the data in an array such that the number of elements and the dimensionality remain the same, but the shape differs. It accepts a new shape whose product must equal the array's current number of elements. In other words, the dimension stays the same but the extents change in limited ways. For example, given the array,

$$A = \left[\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix} \quad \begin{bmatrix} 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \end{bmatrix} \right]$$

the reshape operation,

```
A.reshape(extents[4][3][2]);
```

alters the array as follows:

$$A = \begin{bmatrix} \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} & \begin{bmatrix} 6 & 7 \\ 8 & 9 \\ 10 & 11 \end{bmatrix} & \begin{bmatrix} 12 & 13 \\ 14 & 15 \\ 16 & 17 \end{bmatrix} & \begin{bmatrix} 18 & 19 \\ 20 & 21 \\ 22 & 23 \end{bmatrix} \end{bmatrix}$$

The `multi_array_ref` and `const_multi_array_ref` wrapper classes

MultiArray provides two class templates that model the MultiArray concept and wrap native arrays. The `multi_array_ref` class template wraps a pointer to a mutable, contiguous block of elements. It takes two of the same template parameters as `multi_array`, the type of contained elements, `Element`, and the dimensionality, `NumDims`. Each `multi_array_ref` constructor takes a pointer to the wrapped data as its first parameter. Otherwise, the same means of specifying array shape are accepted and each constructor takes an optional storage order parameter. The `multi_array_ref` interface supports most of the same features as `multi_array`, including iterators, subarrays, views, reshaping, and the comparison operators, however, `multi_array_ref` does not support resizing. The `const_multi_array_ref` class template is a variant of `multi_array_ref` that wraps pointers to const data. This distinction is necessary to preserve constness of pointers when wrapped with a MultiArray interface.

The following code shows some examples of using `const_multi_array_ref` and `multi_array_ref` to wrap a native C++ array.

```
double A_data[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }
double const* const_A_data = A_data;

multi_array_ref<double,1> A(A_data, extents[3][3]);
multi_array_ref<double,1> bad_A(const_A_data, extents[3][3]); // Compile Error!
const_multi_array_ref<double,1> const_A(const_A_data, extents[3][3]); // OK.
```

Library Implementation

MultiArray strives to provide a clean and clear interface for manipulating array abstractions in useful and nontrivial ways. The library uses some advanced C++ techniques in order to achieve these goals. For example, template meta-programming is used to ensure static safety and reasonable performance. We now describe some of the more interesting and important techniques used to realize the MultiArray library components.

Accessing subarrays and elements using bracket notation

MultiArray's array interfaces support addressing values using the same bracket-based notation as a native array. The basic technique, illustrated in [3] uses partial specialization to special-case the indexing of a one-dimensional array. For example, consider the following simplified example of this technique:

```

template <class T, int N>
struct array {
    array<T,N-1> operator[](int idx);
    // ...
}

template <class T>
struct array<T,1> {
    T& operator[](int idx);
    // ...
}

```

Calling `operator[]()` on a one-dimensional array returns an element reference, `T&`, but in the case of a higher dimensional array, the result is an array of one less dimension. This partial specialization technique is effective, but compiler support for it is not yet ubiquitous. `MultiArray` instead uses full template specialization, a better supported part of C++, to simulate the same functionality.

The technique is outlined in the following code:

```

template<class T, int N>
struct access_n {
    array<T,N-1> operator[](int idx);
    // ...
};

template<typename T>
struct access_one {
    T& operator[](int idx);
    // ...
};

struct choose_n {
    template <typename T, int N>
    struct bind { typedef access_n<T,N> type; };
};

struct choose_one {
    template <typename T, int N>
    struct bind { typedef access_one<T> type; };
};

template <int N>
struct helper { typedef choose_n choice; };

template <>
struct helper<1> { typedef choose_one choice; };

template <typename T, int N>
struct generator {
    typedef typename helper<N>::choice choice_t;
    typedef typename choice_t::template bind<T,N>::type type;
};

```

```
template <class T, int N>
struct array : public generator<T,N>::type {
    // ...
}
```

The two implementations of `operator[]()` are factored to two unrelated classes, `access_one` for the one-dimensional case and `access_n` for the multi-dimensional case. The class `array` then uses the `generator` meta-program to choose which of the two classes to inherit from. This technique involves no partial specialization. The `helper` structure is fully specialized on its `N` parameter. The `generator` structure encapsulates the somewhat complex indirection used to choose which class to inherit. This extra effort results in an implementation that works on more compilers than the simpler partial specialization based method.

Avoiding partial specialization with `const_multi_array_ref`

The `const_multi_array_ref` class circumvents the use of partial specialization. In order to support wrapping const-qualified pointers, a separate implementation of the ref class is needed. Partial specialization enables this class to be implemented as a specialization of `multi_array_ref` for types of the form `const T`: rather than write

```
const_multi_array_ref<double,2>
```

the syntax would be

```
multi_array_ref<const double, 2>
```

Since some compilers cannot support this technique, a separate class name was preferred.

Implementation-level code reuse

MultiArray implements many array types: subarrays, views, `multi_array`, `multi_array_ref`, and `const_multi_array_ref`. All these arrays share features in common, including access operations, iterator expressions, and view creation. In order to minimize code duplication and its associated maintenance headaches, much of the array implementation is implemented as a hierarchy of classes, each of which is responsible for a cohesive piece of array functionality. Figure 2 illustrates dependencies between implementation components.

Many types in the library require access to the same type names. These names were encapsulated in the class `multi_array_base`. The two classes `value_accessor_n` and `value_accessor_one` encapsulate the machinery for mapping from indices to values. These pieces are separated from other array operations because they are also used to implement iterators. `multi_array_impl_base` encapsulates most other array operations including view creation, indexing, generating strides, and so on. These pieces are used by all the array classes in the library. The subarray classes, view classes, and `multi_array_ref` have both mutable and const versions to account for differences in their interfaces. In all three cases, the mutable version of the array class is implemented in terms of the const version, leading to more modular

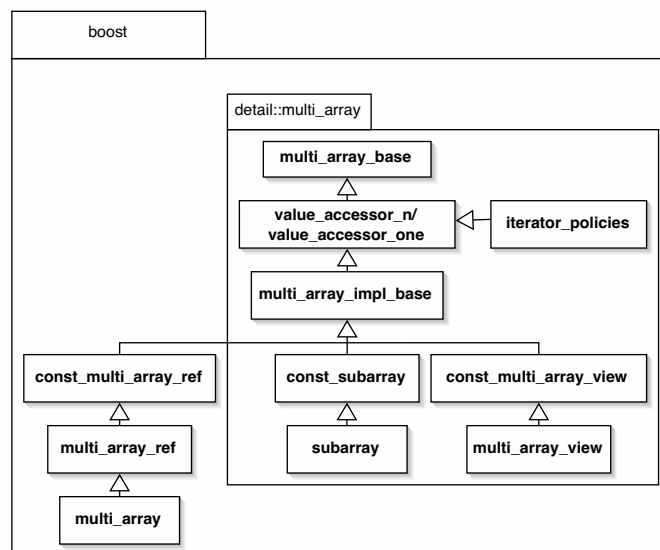


Figure 2. Class hierarchy of the MultiArray library implementation

and maintainable implementation. Finally, the **multi_array** class is implemented in terms of **multi_array_ref**. The difference between the two is **multi_array**'s support for memory management and resizing.

Extent and Index Generators

MultiArray provides two generator objects, **extents** and **indices**, to respectively specify array shape and view specifications. They use **operator[]()** to collect ranges and integers for constructors and calls to **operator[]()** on arrays. The implementation of these objects relies on member templates for functionality and compiler inlining for performance.

The following code shows the general technique used to implement the **extents** and **indices** generator objects.

```

template <int Size>
struct generator {
    int data[Size == 0 ? 1 : Size];

    generator() { }
    generator(generator<Size-1> const& other, int datum) {
        for (int i = 0; i != Size-1; ++i)
            data[i] = other.data[i];
    }
};
  
```

```

    data[Size-1] = datum;
}

generator<Size+1> operator[](int datum) {
    return generator<Size+1>(*this,datum);
}
};

```

The two objects, **extents** and **indices** are equivalent to objects of type **generator<0>**. When **operator[]()** is applied to **generator<N>**, it returns a new object of type **generator<N+1>** that contains a list of all previous elements plus the parameter to the call. Experiments with optimizing C++ compilers show that the intermediate objects disappear and the resulting code constructs only one object and assigns to it all the values. The implementation of **indices** is a little more complicated than the example above because it must separately count the ranges and integers passed to it.

Simulating reference types with classes

The Standard Library containers all have associated reference types and value types. The **std::vector** class template, for instance, returns its reference type as the result of **operator[]()**. Through this reference, user code can read and update the contents of the vector. In general, the reference types of Standard Library containers are true C++ references; the implementation of the containers mostly specify

```
typedef T& reference
```

where **T** is the type of the contained values. MultiArray cannot use true C++ references. In their place, the result of **operator[]()** is an object of type **subarray<N>::type**, a proxy class designed to exhibit reference semantics. Rather than maintain its own copies of array state, including strides, shape, and index bases, a newly constructed subarray contains pointers to the data held within the array used to construct it. Any subsequent use of the subarray manipulates the referenced data. Modifying a subarray's elements modifies the parent array. Assigning one subarray from another does not change which array the subarray refers to. Rather, it results in a deep copy of elements of one array into the other. The MultiArray view classes are designed with the same sort of reference semantics.

Unfortunately, certain aspects of C++ undermine this facade. The most difficult of these traits is that a temporary object cannot be passed to a function that takes the parameter by reference. Consider the following code example:

```

template <class Array>
void mutate_array(Array&);
//...
mutate_array(A[7]); // Error: Passing temporary to a reference

```

Here, a template function **mutate_array()** takes its parameter by reference. The call site passes a subarray to the function. This code is erroneous because the result of **A[7]** is a temporary subarray object. C++ has this rule in order to prevent what might be a difficult to detect programming error, namely mutating an object whose lifetime does not exceed the

function call site. An insidious side effect of this is an inability to simulate reference types using C++ objects.

Related Work

The design of the MultiArray library was influenced significantly by several prior projects related to array programming in C++. Blitz++ [17] is a high-performance library for array computations. Many techniques for developing high-performance scientific C++ libraries first appeared in this library and some of the techniques and abstractions used in MultiArray, including storage order, were borrowed from it. MultiArray differs from Blitz++ in that greater emphasis is placed on developing an interface that mimics C++ native arrays. Additionally, MultiArray places greater emphasis on portability among compiler implementations and compatibility with the C++ Standard Library containers.

In [3], Bavestrelli develops a rudimentary array class that supports the use of bracket notation to access elements. MultiArray adds, beyond this functionality, techniques for building array views and a conceptual interface upon which generic array libraries can be based.

Dietmar Khl developed an Array Traits library, formerly distributed with Boost, that supports creating iterators for native C++ arrays much like MultiArray array iterators. Nicolai Josuttis developed the Boost Array Library, which wraps fixed-length one-dimensional native C++ arrays with an interface modeled after the Standard Library containers.

The A++/P++ [12] and POOMA [7] libraries implement arrays for local and distributed computation. These libraries emphasize abstracting array distribution while maintaining high-performance. The A++/P++ library uses the same interface to implement a simple local (A++) array library and its parallel equivalent (P++). POOMA uses expression template techniques [16] to optimize array operations at compile time and distribute computation across machines. Neither of these libraries support using brackets to access subarrays as native C++ arrays do, but both place great emphasis on performance.

The Standard Library supplies a component, `std::valarray`, for building array-like mathematical abstractions. This class is meant to be a building block for higher-level array abstractions, but few, if any, have been developed.

Future work

Some plans to enhance the MultiArray library are already proceeding. Element-wise arithmetic operations will be added for element types that support them. The library's development has centered on its user interface and interoperability with the C++ Standard Library. Preliminary performance testing ensured that the interface design does not irrevocably undermine performance. Further performance testing and tuning is forthcoming. Research is underway to apply MultiArray's interface philosophy in the arena of data-parallel computation.

Conclusion

The MultiArray library augments the C++ Standard Library with support for multi-dimensional arrays. In addition to the expected indexing functionality of arrays, MultiArray adds support for views and subarrays by which a portion of an existing array can be manipulated as a first-class array itself. The MultiArray concept formalizes a generic interface to arrays against which generic algorithms that can operate on any MultiArray array types can be implemented. The library leverages advanced C++ techniques to present a friendly, effective, and versatile interface to the application developer.

Acknowledgements

This work was supported by a grant from the Lilly Endowment and NSF grants ACI-0219884 and EIA-0131354.

REFERENCES

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra package for high-performance computers. In *Proceedings of Supercomputing '90*, pages 1–10. IEEE Press, 1990.
2. M. H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley, 1999.
3. G. Bavestrelli. A class template for n-dimensional generic resizable arrays. *C/C++ Users Journal*, December 2000.
4. I. O. for Standardization (ISO). *ISO/IEC Final Draft International Standard 14882: Programming Language C++*. 1 rue de Varembe, Case postale 56, CH-1211 Genève 20, Switzerland, 1998.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
6. N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, 1999.
7. S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T. J. Williams. Array design and expression evaluation in POOMA II. In D. Caromel, R. R. Oldehoeft, and M. Tholburn, editors, *ISCOPE*. Advanced Computing Laboratory, LANL, 1998.
8. C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.
9. B. McNamara and Y. Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming*, October 2000.
10. D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 2nd edition, 2001.
11. N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.
12. R. Parsons and D. Quinlan. A++/P++ array classes for architecture independent finite difference computations. In *Object-Oriented Numerics Conference (OONSKI)*, pages 408–418, April 24–27, 1994.
13. J. Siek. Collection concept. <http://www.boost.org/libs/utility/Collection.html>, 2000.
14. J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, October 2000.
15. A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.
16. T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in *C++ Gems*, ed. Stanley Lippman.

-
17. T. L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, volume 1505 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.