Accredited Standards Committee
X3, INFORMATION PROCESSING SYSTEMS

Doc No: ???
Date: 5/27/93
Project: Programming Language C++
Ref Doc: X3J16-93-0073/WG21-N0280
Reply To: Kent G. Budge
kgbudge@sherpa.sandia.gov

Proposal for a Numerical Array Library (Revised)

1. Introduction

One of the most widely recognized inadequacies of C is its low-level treatment of arrays. Arrays are not first-class objects in C; an array name in an expression almost always decays into a pointer to the underlying type. This is unfortunate, especially since an increasing number of high-performance computers are optimized for calculations involving arrays of numbers. On such machines, `double`[] may be regarded as an intrinsic data type comparable to `double` or `int` and quite distinct from `double*`.

This weakness of C is acknowledged in the ARM [4], where it is suggested that the inadequacies of the C array can be overcome in C++ by wrapping it in a class that supplies dynamic memory management, bounds checking, operator syntax, and other useful features. Such “smart arrays” can in fact supply the same functionality as the first-class arrays found in other high-level, general-purpose programming languages. Unfortunately, they are expensive in both time and memory and make poor use of advanced floating-point architectures [2], [3].

Is there a better solution? The most obvious solution is to make arrays first-class objects and add the functionality mentioned in the previous paragraph. However, this would destroy C compatibility and significantly alter the C++ language. Major conflicts with existing practice would seem inevitable.

I propose instead that numerical array classes be adopted as part of the C++ standard library. These classes will have the functionality appropriate for the intrinsic arrays found on most high-performance computers, and the compilers written for these computers will be free to implement them as built-in classes. On other platforms, these classes may be defined normally, and will provide users with basic array functionality without imposing an excessive burden on the implementor.

2. Rationale

Numerical code makes frequent use of array-like data objects. Examples include *fields* (such as the temperature distribution in an internal combustion engine), *matrices* (used in the representation of simultaneous linear equations), or *datasets* (such as climatological data for a single weather station). All these are specializations of the concept of an ordered set of data.

The C++ programming language provides mechanisms for writing extremely expressive, modular, and reusable code for such applications. For example, a computational physicist such as myself makes frequent use of the expression

---

* *Operating under the procedures of the American National Standards Institute (ANSI)*

$$\vec{f} = \vec{\nabla} \bullet \mathbf{T} + \vec{b}$$

which is the momentum equation for a continuous medium (such as a fluid or solid). The objects $\vec{f}$ and $\vec{b}$ are vector fields; $\mathbf{T}$ is a tensor field; and $\vec{\nabla}$ is the gradient operator (here used to calculate a divergence), which has many of the properties of a vector field. I can represent each of these kinds of object by a suitable class and provide overloaded operators so that the computer code becomes

```
f = mesh->Grad() * T + b;
```

This code is clear, expressive, and modular. The underlying classes should in principle be highly reusable, although experience to date is limited by the relative youthfulness of C++ in the numerical world.

Unfortunately, array computations are inefficient in C++ as presently defined in the working draft. This is true even for low-level array code.

2.1 Aliasing Ambiguities

Consider the following translation unit:

```
#include <stddef.h>

void subroutine(const double *op1, const double *op2,
                double *sum, double *diff, size_t N){
  for (register i=0; i<N; i++){
    sum[i] = op1[i] + op2[i];
    diff[i] = op1[i] - op2[i];
  }
}
```

The intent of this subroutine is that it evaluates the sum and difference of two arrays, storing the result in two other arrays. Ideally, the global register allocation scheme should load `op1[i]` and `op2[i]` into registers and avoid repeating memory fetches for the second statement in the loop, but it cannot safely do so. There is no guarantee that `op1`, `op2`, and `sum` do not point to the same array. In other words, we have encountered an *aliasing ambiguity*, a problem well-known to implementors of optimizing C compilers. Note that the presence of `const` qualifiers in the argument list does not change this, since the conversion to `const` is a trivial conversion.

The performance penalty incurred by aliasing ambiguities varies considerably with architecture and application. On RISC architectures, it can be up to a factor of two. On certain high-performance architectures, the penalty can be even higher. In particular, many supercomputers suffer from a form of aliasing ambiguity known as *vector recursion*. This occurs because the processor has vector registers, which are capable of storing and operating on entire arrays of data. The processor cannot effectively use these registers unless it can load them from memory at the beginning of an expression evaluation. An expression such as `sum[i] = op1[i] + op2[i]` in the prior example cannot be safely evaluated using vector registers and operations, since the compiler cannot determine whether `sum[i]` is an alias for (for example) `op1[i+1]` or `op2[i+1]`, which would be loaded before `sum[i]` is evaluated. Instead, the processor must evaluate this expression using scalar registers and operations, which are a factor *~40-80* slower than their vector counterparts.

Some languages, such as FORTRAN-77, legislate away aliasing ambiguities. This is not an option in C or C++, where restrictions on pointer arguments would violate the spirit of the languages. Even the adoption of a `restrict` pointer qualifier [5] has proven unacceptable to the committee. However, one can eliminate aliasing ambiguities by using the idiom

```
#include <stddef.h>
#include <stdlib.h>

void subroutine(const double *op1, const double *op2,
                double *&sum, double *&diff, size_t N){
  sum = new double[N];
  diff = new double[N];
```

```
  for (register i=0; i<N; i++){
    sum[i] = op1[i] + op2[i];
    diff[i] = op1[i] - op2[i];
  }
}
```

The pointer returned by the `new` operator is guaranteed by the working draft to be free from aliases. (This is implied by the requirement that `new` must return a pointer to a distinct object.) Thus, a compiler can assume that storing a value in `sum` will not invalidate any register copy of values from `op1` or `op2`.

Unfortunately, the above idiom, while permitting a higher degree of optimization, is somewhat unsafe because of the potential for dangling pointers. The C++ language encourages a rather Puritan style of programming in which there are no naked pointers; they should be clothed in a class definition. Consider the following translation unit:

```
#include <stddef.h>
#include <stdlib.h>
#include <assert.h>
#include <numarray>

void subroutine(const dblarray& op1, const dblarray& op2,
                dblarray& sum, dblarray& diff){
  const size_t N = op1.length();
  assert(N == op2.length());// conformance check
  sum = dblarray(N);
  diff = dblarray(N);
  for (register i=0; i<N; i++){
    sum[i] = op1[i] + op2[i];
    diff[i] = op1[i] - op2[i];
  }
}
```

I include a header, `<numarray>`, which declares a smart array class, `dblarray`. This class replaces each pointer to double. Not only are there no naked pointers, but I can slip in a conformance check as well, since the size of each array is now riding around with the array itself. The constructor for `dblarray` uses `operator new` to allocate storage, and if the constructor and the subscript operator are simple inline functions, the compiler has the information it needs to deduce that no aliases exist. The classes described in this proposal support this idiom.

2.2 The Problem of Temporaries

The difficulty discussed in the previous section is apparent at a very low level of programming. In this section, I discuss a problem that is not apparent until one attempts to overload operators for array-like objects.

Suppose we overload the assignment and arithmetic operators for the class `dblarray`. One can then write quite complicated array calculations as simple expressions, such as:

```
y = a + x*(b + x*c);
```

This code is very expressive, but also very inefficient with current compilers. The generated code calculates the entire set of values resulting from each operation before going on to the next operation in the parse tree. In effect (assuming all operations are inlined) the expansion of this single line of code is

```
const size_t N = x.N;
tmp1.data = new double[N];
for (i=0; i<N; i++) tmp1.data[i] = x.data[i] * c.data[i];
tmp2.data = new double[N];
for (i=0; i<N; i++) tmp2.data[i] = b.data[i] + tmp1.data[i];
tmp3.data = new double[N];
for (i=0; i<N; i++) tmp3.data[i] = x.data[i] * tmp2.data[i];
tmp4.data = new double[N];
for (i=0; i<N; i++) tmp4.data[i] = a.data[i] + tmp3.data[i];
```

```
for (i=0; i<N; i++) y.data[i] = tmp4.data[i];
delete[] tmp4.data;
delete[] tmp3.data;
delete[] tmp2.data;
delete[] tmp1.data;
```

I have left out some bookkeeping details, which may be found in [2] and [3]. There are several serious inefficiencies in this expansion:

1. Large amounts of memory are allocated to store intermediate results.
2. Intermediate results are stored and then almost immediately fetched again.
3. Loops are separated, which hinders efficient global register allocation. For example, `x.data[i]` will generally have to be loaded twice.
4. The machinery for identical loops is duplicated.

The ideal expansion would be

```
for (i=0; i<N; i++)
  y.data[i] = a.data[i] + x.data[i] * (b.data[i] + x.data[i] * c.data[i]);
```

which yields exactly the same result for y but avoids all these difficulties.

The optimization algorithms needed to transform the first expansion into the second are possible, but they are difficult to implement for the general case. My proposed solution is to standardize an array class which a compiler can recognize and for which it can specifically provide these optimizations.

3. Proposal

I propose that the following material be incorporated into the working draft and rationale:

3.1 Mathematical Arrays

The standard library includes the header `<numarray>`, which defines three classes, `dblarray`, `fltarray`, and `intarray`. These classes support mathematical operations on arrays of numbers.

❑ These classes may be implemented as built-in types introduced into the user name space through a typedef. For example, the header may be a file containing the text

```
#if !defined(__COMPILER_DEBUG_FLAG_ON)

typedef __builtin_dblarray dblarray;
typedef __builtin_fltarray fltarray;
typedef __builtin_intarray intarray;

#else

class dblarray {
  /*...normal class definition with debugging support...*/
};
class fltarray {
  /*...normal class definition with debugging support...*/
};
class intarray {
  /*...normal class definition with debugging support...*/
};
```

```
#endif
```

However, nothing prevents these types from being implemented solely as ordinary classes.

The advantage of implementing these classes as built-in types is that they have optimization properties that are not immediately apparent from the class definition. A partial alternative to built-in types is to insert `#pragma` directives in the class declaration to inform the compiler of their optimization properties.

❑ Because these classes are intended to be highly optimized and/or have built-in compiler support, it was thought inappropriate to mandate that they be specializations of a template. This suggests that the correct idiom for building (for example) a complex array is the object-of-arrays idiom rather than the array-of-objects idiom. That is, one writes classes like

```
class complex_array {
  dblarray re, im;
  /* ... */
};
```

rather than

```
class complex_array {
  complex *data;
  /* ... */
};
```

The appropriate way to use templates is then

```
class complex_template<class T> {
  T re, im;
  /* ... */
};
typedef complex_template<double> complex;
typedef complex_template<dblarray> complex_array;
```

This idiom does make it difficult to write an efficient subscript operator for modifying individual elements of the array. On the other hand, it facilities efficient access to the real and imaginary parts of the array as a whole, which may be equally important. One can still write efficient access functions to replace the subscript operator for low-level programming.

If one must have a subscript operator, and one is willing to give up templates, an alternative is the “object-of-array” idiom:

```
class complex_array {
  dblarray data;
  /* ... */
}
```

The complex class is a POD (Plain Old Data) composed of two doubles. One can therefore store a set of such objects in a double-length `dblarray`. The subscript operator can then be efficiently implemented, and the real and imaginary parts can be separated out of the array using the stride-based `gather` and `scatter` operations (see 3.1.1.31 on page 31). The object-of-array idiom is also useful when the array must be passed into a FORTRAN program using `extern "FORTRAN"` linkage.

3.1.1 Public Interface

The classes `dblarray`, `fltarray`, and `intarray` have the following public interface:

```
class dblarray;                          // An array of double values
class fltarray;                          // An array of float values
class intarray;                          // An array of integer values

class dblarray {
  public:

              /**** Constructors, destructors, assigns ****/

    dblarray(void);
    dblarray(size_t);
    dblarray(size_t, double);
    dblarray(size_t, const double*);

    dblarray(const dblarray&);
    dblarray(const fltarray&);
    dblarray(const intarray&);

    ~dblarray(void);

    dblarray& operator=(const dblarray&);
    dblarray& operator=(const fltarray&);
    dblarray& operator=(const intarray&);

              /**** Operator overloads ****/

// Access functions

    size_t length(void) const;
    double operator[](size_t) const;
    double& operator[](size_t);

    operator double*(void);
    operator const double*(void) const;

// Unary operators

    dblarray operator+(void) const;
    dblarray operator-(void) const;

// Operators with scalar constants

    friend dblarray operator*(const dblarray&, double);
    friend dblarray operator*(double, const dblarray&);
    dblarray& operator*=(double);
    friend dblarray operator/(const dblarray&, double);
    friend dblarray operator/(double, const dblarray&);
    dblarray& operator/=(double);
    friend dblarray operator+(const dblarray&, double);
    friend dblarray operator+(double, const dblarray&);
    dblarray& operator+=(double);
    friend dblarray operator-(const dblarray&, double);
    friend dblarray operator-(double, const dblarray&);
    dblarray& operator-=(double);

// Operators with other dblarray

    friend dblarray  operator*(const dblarray&, const dblarray&);
    dblarray& operator*=(const dblarray& ab);
    friend dblarray  operator/(const dblarray&, const dblarray&);
    dblarray& operator/=(const dblarray& ab);
    friend dblarray  operator+(const dblarray&, const dblarray&);
```

```
    dblarray& operator+=(const dblarray& ab);
    friend dblarray  operator-(const dblarray&, const dblarray&);
    dblarray& operator-=(const dblarray& ab);

// operators returning intarray

    friend intarray operator==(const dblarray&, double);
    friend intarray operator==(double, const dblarray&);
    friend intarray operator==(const dblarray&, const dblarray&);
    friend intarray operator!=(const dblarray&, double);
    friend intarray operator!=(double, const dblarray&);
    friend intarray operator!=(const dblarray&, const dblarray&);
    friend intarray operator<(const dblarray&, double);
    friend intarray operator<(double, const dblarray&);
    friend intarray operator<(const dblarray&, const dblarray&);
    friend intarray operator>(const dblarray&, double);
    friend intarray operator>(double, const dblarray&);
    friend intarray operator>(const dblarray&, const dblarray&);
    friend intarray operator<=(const dblarray&, double);
    friend intarray operator<=(double, const dblarray&);
    friend intarray operator<=(const dblarray&, const dblarray&);
    friend intarray operator>=(const dblarray&, double);
    friend intarray operator>=(double, const dblarray&);
    friend intarray operator>=(const dblarray&, const dblarray&);

// Methods

    friend dblarray where(const intarray&, const dblarray&, const dblarray&);
    friend dblarray where(const intarray&, const dblarray&, double);
    friend dblarray where(const intarray&, double, const dblarray&);
    friend dblarray where(const intarray&, double, double);
    double sum(void) const;
    void fill(double);

    friend dblarray acos(const dblarray&);
    friend dblarray asin(const dblarray&);
    friend dblarray atan(const dblarray&);
    friend dblarray atan2(const dblarray& y, double x);
    friend dblarray atan2(double y, const dblarray& x);
    friend dblarray atan2(const dblarray& y, const dblarray& x);
    friend intarray ceil(const dblarray&);
    friend dblarray cos(const dblarray&);
    friend dblarray cosh(const dblarray&);
    friend dblarray exp(const dblarray&);
    friend dblarray fabs(const dblarray&);
    friend intarray floor(const dblarray&);
    friend dblarray frexp(const dblarray&, intarray&)
    friend dblarray fmod(const dblarray&, double);
    friend dblarray fmod(double, const dblarray&);
    friend dblarray fmod(const dblarray&, const dblarray&);
    friend dblarray ldexp(const dblarray&, const intarray&);
    friend dblarray log(const dblarray&);
    friend dblarray log10(const dblarray&);
    friend dblarray modf(const dblarray&, dblarray&);
    friend dblarray pow(const dblarray&, double);
    friend dblarray pow(double, const dblarray&);
    friend dblarray pow(const dblarray&, const dblarray&);
    friend dblarray sin(const dblarray&);
    friend fltarray sngl(const dblarray&);
    friend dblarray sinh(const dblarray&);
    friend dblarray sqrt(const dblarray&);
    friend dblarray tan(const dblarray&);
```

```
    friend dblarray tanh(const dblarray&);

// FORTRAN-like functions

    friend dblarray dim(const dblarray&, double);
    friend dblarray dim(double, const dblarray&);
    friend dblarray dim(const dblarray&, const dblarray&);
    friend dblarray min(double, const dblarray&);
    friend dblarray max(double, const dblarray&);
    friend dblarray min(const dblarray&, double);
    friend dblarray max(const dblarray&, double);
    friend dblarray min(const dblarray&, const dblarray&);
    friend dblarray max(const dblarray&, const dblarray&);
    friend double min(const dblarray&);
    friend double max(const dblarray&);
    friend dblarray sign(double, const dblarray&);
    friend dblarray sign(const dblarray&, double);
    friend dblarray sign(const dblarray&, const dblarray&);

// Topological methods

    dblarray shift(int n) const;

    dblarray& assemble(const dblarray&, const intarray&, const intarray&);
    dblarray& assemble(const dblarray&, const intarray&);
    dblarray& assemble(double, const intarray&);
    dblarray& scatter(const dblarray&, const intarray&, const intarray&);
    dblarray& scatter(const dblarray&, const intarray&);
    dblarray& scatter(double, const intarray&);
    dblarray  gather(const intarray&) const;
    dblarray& scatter(const dblarray&, size_t, size_t, size_t);
    dblarray& scatter(const dblarray&, size_t, size_t);
    dblarray  gather(size_t, size_t, size_t) const;
    dblarray  gather(size_t, size_t) const;

// I/O

    friend ostream& operator<<(ostream&, const volatile dblarray&);
    friend istream& operator>>(istream&, dblarray&);

// miscellaneous

    void free(void);

  private:

    /* ... implementation-dependent ... */
};

class fltarray {
  public:

             /**** Constructors, destructors, assigns ****/

    fltarray(void);
    fltarray(size_t);
    fltarray(size_t, float);
    fltarray(size_t, const float*);

    fltarray(const fltarray&);
    fltarray(const intarray&);
```

```
    ~fltarray(void);

    fltarray& operator=(const fltarray&);
    fltarray& operator=(const intarray&);

             /**** Operator overloads ****/

// Access functions

    size_t length(void) const;
    float operator[](size_t) const;
    float& operator[](size_t);

    operator float*(void);
    operator const float*(void) const;

// Unary operators

    fltarray operator+(void) const;
    fltarray operator-(void) const;

// Operators with scalar constants

    friend fltarray operator*(const fltarray&, float);
    friend fltarray operator*(float, const fltarray&);
    fltarray& operator*=(float);
    friend fltarray operator/(const fltarray&, float);
    friend fltarray operator/(float, const fltarray&);
    fltarray& operator/=(float);
    friend fltarray operator+(const fltarray&, float);
    friend fltarray operator+(float, const fltarray&);
    fltarray& operator+=(float);
    friend fltarray operator-(const fltarray&, float);
    friend fltarray operator-(float, const fltarray&);
    fltarray& operator-=(float);

// Operators with other fltarray

    friend fltarray  operator*(const fltarray&, const fltarray&);
    fltarray& operator*=(const fltarray& ab);
    friend fltarray  operator/(const fltarray&, const fltarray&);
    fltarray& operator/=(const fltarray& ab);
    friend fltarray  operator+(const fltarray&, const fltarray&);
    fltarray& operator+=(const fltarray& ab);
    friend fltarray  operator-(const fltarray&, const fltarray&);
    fltarray& operator-=(const fltarray& ab);

// operators returning intarray

    friend intarray operator==(const fltarray&, float);
    friend intarray operator==(float, const fltarray&);
    friend intarray operator==(const fltarray&, const fltarray&);
    friend intarray operator!=(const fltarray&, float);
    friend intarray operator!=(float, const fltarray&);
    friend intarray operator!=(const fltarray&, const fltarray&);
    friend intarray operator<(const fltarray&, float);
    friend intarray operator<(float, const fltarray&);
    friend intarray operator<(const fltarray&, const fltarray&);
    friend intarray operator>(const fltarray&, float);
    friend intarray operator>(float, const fltarray&);
    friend intarray operator>(const fltarray&, const fltarray&);
    friend intarray operator<=(const fltarray&, float);
```

```
    friend intarray operator<=(float, const fltarray&);
    friend intarray operator<=(const fltarray&, const fltarray&);
    friend intarray operator>=(const fltarray&, float);
    friend intarray operator>=(float, const fltarray&);
    friend intarray operator>=(const fltarray&, const fltarray&);

// Methods

    friend fltarray where(const intarray&, const fltarray&, const fltarray&);
    friend fltarray where(const intarray&, const fltarray&, float);
    friend fltarray where(const intarray&, float, const fltarray&);
    friend fltarray where(const intarray&, float, float);
    float sum(void) const;
    void fill(float);

    friend fltarray acos(const fltarray&);
    friend fltarray asin(const fltarray&);
    friend fltarray atan(const fltarray&);
    friend fltarray atan2(const fltarray& y, float x);
    friend fltarray atan2(float y, const fltarray& x);
    friend fltarray atan2(const fltarray& y, const fltarray& x);
    friend intarray ceil(const fltarray&);
    friend fltarray cos(const fltarray&);
    friend fltarray cosh(const fltarray&);
    friend fltarray exp(const fltarray&);
    friend fltarray fabs(const fltarray&);
    friend intarray floor(const fltarray&);
    friend fltarray frexp(const fltarray&, intarray&)
    friend fltarray fmod(const fltarray&, float);
    friend fltarray fmod(float, const fltarray&);
    friend fltarray fmod(const fltarray&, const fltarray&);
    friend fltarray ldexp(const fltarray&, const intarray&);
    friend fltarray log(const fltarray&);
    friend fltarray log10(const fltarray&);
    friend fltarray modf(const fltarray&, fltarray&);
    friend fltarray pow(const fltarray&, float);
    friend fltarray pow(float, const fltarray&);
    friend fltarray pow(const fltarray&, const fltarray&);
    friend fltarray sin(const fltarray&);
    friend fltarray sinh(const fltarray&);
    friend fltarray sqrt(const fltarray&);
    friend fltarray tan(const fltarray&);
    friend fltarray tanh(const fltarray&);

// FORTRAN-like functions

    friend fltarray dim(const fltarray&, float);
    friend fltarray dim(float, const fltarray&);
    friend fltarray dim(const fltarray&, const fltarray&);
    friend fltarray min(float, const fltarray&);
    friend fltarray max(float, const fltarray&);
    friend fltarray min(const fltarray&, float);
    friend fltarray max(const fltarray&, float);
    friend fltarray min(const fltarray&, const fltarray&);
    friend fltarray max(const fltarray&, const fltarray&);
    friend float min(const fltarray&);
    friend float max(const fltarray&);
    friend fltarray sign(float, const fltarray&);
    friend fltarray sign(const fltarray&, float);
    friend fltarray sign(const fltarray&, const fltarray&);

// Topological methods
```

```
    fltarray shift(int n) const;

    fltarray& assemble(const fltarray&, const intarray&, const intarray&);
    fltarray& assemble(const fltarray&, const intarray&);
    fltarray& assemble(float, const intarray&);
    fltarray& scatter(const fltarray&, const intarray&, const intarray&);
    fltarray& scatter(const fltarray&, const intarray&);
    fltarray& scatter(float, const intarray&);
    fltarray  gather(const intarray&) const;
    fltarray& scatter(const fltarray&, size_t, size_t, size_t);
    fltarray& scatter(const fltarray&, size_t, size_t);
    fltarray  gather(size_t, size_t, size_t) const;
    fltarray  gather(size_t, size_t) const;

// I/O

    friend ostream& operator<<(ostream&, const volatile fltarray&);
    friend istream& operator>>(istream&, fltarray&);

// miscellaneous

    void free(void);

  private:

    /* ... implementation-dependent ... */
};


class intarray {
  public:

              /**** Constructors, destructors, assigns ****/

    intarray(void);
    intarray(size_t);
    intarray(size_t size, int fill);
    intarray(size_t, const int*);

    intarray(const intarray&);

    ~intarray(void);

    intarray& operator=(const intarray&);

              /**** Operator overloads ****/

// Access functions

    size_t length(void) const;
    int operator[](size_t) const;
    int& operator[](size_t);
    operator const int *(void) const;
    operator int *(void);

// Unary operators

    intarray operator+(void) const;
    intarray operator-(void) const;
    intarray operator~(void) const;
    intarray operator!(void) const;
```

```
// Operators with scalar constants

    friend intarray operator*(const intarray&, int);
    friend intarray operator*(int, const intarray&);
    intarray& operator*=(int);
    friend intarray operator/(const intarray&, int);
    friend intarray operator/(int, const intarray&);
    intarray& operator/=(int);
    friend intarray operator%(const intarray&, int);
    friend intarray operator%(int, const intarray&);
    intarray& operator%=(int);
    friend intarray operator+(const intarray&, int);
    friend intarray operator+(int, const intarray&);
    intarray& operator+=(int);
    friend intarray operator-(const intarray&, int);
    friend intarray operator-(int, const intarray&);
    intarray& operator-=(int);
    friend intarray operator^(const intarray&, int);
    friend intarray operator^(int, const intarray&);
    intarray& operator^=(int);
    friend intarray operator&(const intarray&, int);
    friend intarray operator&(int, const intarray&);
    intarray& operator&=(int);
    friend intarray operator|(const intarray&, int);
    friend intarray operator|(int, const intarray&);
    intarray& operator|=(int);
    friend intarray operator<(const intarray&, int);
    friend intarray operator<(int, const intarray&);
    friend intarray operator>(const intarray&, int);
    friend intarray operator>(int, const intarray&);
    friend intarray operator<=(const intarray&, int);
    friend intarray operator<=(int, const intarray&);
    friend intarray operator>=(const intarray&, int);
    friend intarray operator>=(int, const intarray&);
    friend intarray operator<<(const intarray&, int);
    friend intarray operator<<(int, const intarray&);
    intarray& operator<<=(int);
    friend intarray operator>>(const intarray&, int);
    friend intarray operator>>(int, const intarray&);
    intarray& operator>>=(int);
    friend intarray operator&&(const intarray&, int);
    friend intarray operator&&(int, const intarray&);
    friend intarray operator||(const intarray&, int);
    friend intarray operator||(int, const intarray&);
    friend intarray operator==(const intarray&, int);
    friend intarray operator==(int, const intarray&);
    friend intarray operator!=(const intarray&, int);
    friend intarray operator!=(int, const intarray&);

// Operators with other integer fields

    friend intarray  operator*(const intarray&, const intarray&);
    intarray& operator*=(const intarray& ab);
    friend intarray  operator/(const intarray&, const intarray&);
    intarray& operator/=(const intarray& ab);
    friend intarray  operator%(const intarray&, const intarray&);
    intarray& operator%=(const intarray& ab);
    friend intarray  operator+(const intarray&, const intarray&);
    intarray& operator+=(const intarray& ab);
    friend intarray  operator-(const intarray&, const intarray&);
    intarray& operator-=(const intarray& ab);
```

```
    friend intarray operator^(const intarray&, const intarray&);
    intarray& operator^=(const intarray&);
    friend intarray operator|(const intarray&, const intarray&);
    intarray& operator|=(const intarray&);
    friend intarray operator&(const intarray&, const intarray&);
    intarray& operator&=(const intarray&);
    friend intarray operator<<(const intarray&, const intarray&);
    intarray& operator<<=(const intarray&);
    friend intarray operator>>(const intarray&, const intarray&);
    intarray& operator>>=(const intarray&);

    friend intarray operator==(const intarray&, const intarray&);
    friend intarray operator!=(const intarray&, const intarray&);
    friend intarray operator< (const intarray&, const intarray&);
    friend intarray operator> (const intarray&, const intarray&);
    friend intarray operator<=(const intarray&, const intarray&);
    friend intarray operator>=(const intarray&, const intarray&);
    friend intarray operator||(const intarray&, const intarray&);
    friend intarray operator&&(const intarray&, const intarray&);

// Methods

    int sum(void) const;
    friend intarray where(const intarray&, const intarray&, const intarray&);
    friend intarray where(const intarray&, const intarray&, int);
    friend intarray where(const intarray&, int, const intarray&);
    friend intarray where(const intarray&, int, int);
    void fill(int);

    friend intarray abs(const intarray&);
    friend intarray ceil(const dblarray&);
    friend void div(const intarray&, const intarray&, intarray&, intarray&);
    friend intarray floor(const dblarray&);
    void rand(void);

// FORTRAN-like functions

    friend intarray min(int, const intarray&);
    friend intarray max(int, const intarray&);
    friend intarray min(const intarray&, int);
    friend intarray max(const intarray&, int);
    friend intarray min(const intarray&, const intarray&);
    friend intarray max(const intarray&, const intarray&);
    friend int min(const intarray&);
    friend int max(const intarray&);
    friend intarray sign(int, const intarray&);
    friend intarray sign(const intarray&, int);
    friend intarray sign(const intarray&, const intarray&);
    friend intarray dim(const intarray&, int);
    friend intarray dim(int, const intarray&);
    friend intarray dim(const intarray&, const intarray&);

// Perform a shift operation

    intarray shift(int) const;

// Topological operations

    intarray& assemble(const intarray&, const intarray&, const intarray&);
    intarray& assemble(const intarray&, const intarray&);
    intarray& assemble(int, const intarray&);
    intarray& scatter(const intarray&, const intarray&, const intarray&);
```

```
    intarray& scatter(const intarray&, const intarray&);
    intarray& scatter(int, const intarray&);
    intarray  gather(const intarray&) const;
    intarray& scatter(const intarray&, size_t, size_t, size_t);
    intarray& scatter(const intarray&, size_t, size_t);
    intarray  gather(size_t, size_t, size_t) const;
    intarray  gather(size_t, size_t) const;

// I/O

    friend ostream& operator<<(ostream&, const volatile intarray&);
    friend istream& operator>>(istream&, intarray&);

// miscellaneous

    void free(void);

  private:

    /* ... implementation dependent ... */
};
```

❑ The array classes represent one-dimensional arrays, with elements numbered sequentially from zero. They are a representation of the mathematical concept of an ordered set of values. The illusion of higher dimensionality may be produced by the familiar idiom of computed indices.

The intent was to specify array classes that have the minimum functionality necessary to address aliasing ambiguities and the proliferation of temporaries. Thus, the arrays classes are neither matrix classes nor finite-difference/finite-element classes. However, they are expected to be very useful building blocks for designing such classes (either through nesting or private inheritance.)

An implementation is permitted to qualify any of the functions declared in `<numarray>` as `inline`.

❑ An implementor whose compiler has strong inlining capabilities can greatly increase the scope of optimizations in this way. Furthermore, when the array classes are implemented as built-in types, the behavior of the associated built-in array functions and operators is best described as inline.

3.1.1.1 Default constructor

dblarray::dblarray(void)
fltarray::fltarray(void)
intarray::intarray(void)

The array created by the default constructor has zero length until it appears on the left side of an assignment (3.1.1.8) or is passed into a library function through a non-constant reference or non-constant `this` pointer.

❑ A default constructor is essential, since arrays of `dblarray` or `intarray` are likely to prove useful. There must also be a way to change the size of an array after initialization; this is supplied by the semantics of the assignment operator (see 3.1.1.8 on page 16).

The array classes will typically be implemented as smart pointers to blocks of memory containing the data. This memory will normally be allocated off the heap, but an implementation might choose to store array class data on the stack or even in an addressable vector register.

3.1.1.2 Constructor with size argument

dblarray::dblarray(size_t) throw (xnumarray)
fltarray::fltarray(size_t) throw (xnumarray)
intarray::intarray(size_t) throw (xnumarray)

The array created by a constructor having a single `size_t` argument has a length equal to the argument. The values of the elements of the array are undefined.

3.1.1.3 Constructor with size and initializer

dblarray::dblarray(size_t, double) throw (xnumarray)
fltarray::fltarray(size_t, float) throw (xnumarray)
intarray::intarray(size_t, int) throw (xnumarray)

The array created by one of these constructors has a length equal to the first argument. The values of the elements of the array are initialized to the value of the second argument.

3.1.1.4 Constructor from C array

dblarray::dblarray(size_t, const double*) throw (xnumarray)
fltarray::fltarray(size_t, const float*) throw (xnumarray)
intarray::intarray(size_t, const int*) throw (xnumarray)

The array created by one of these constructors has a length equal to the first argument. The values of the elements of the array are initialized with the values pointed to by the second argument. For example,

```
double arr[5] = {3.0, 6.5, 2.0, 4.1, 8.0};
dblarray darr(5, arr);
```

If the value of the first argument is greater than the number of values pointed to by the second argument, the behavior is undefined.

❑ This constructor is the preferred method for converting a C array to a numerical array object.

3.1.1.5 Copy constructor

dblarray::dblarray(const dblarray&) throw (xnumarray)
fltarray::fltarray(const fltarray&) throw (xnumarray)
intarray::intarray(const intarray&) throw (xnumarray)

The array created by this constructor has the same length and element values as the argument.

❑ The copy constructor creates a conceptually distinct array rather than an alias. Implementations in which arrays share storage are possible but must implement a "copy-on-reference" mechanism that includes a reference count and a live reference flag.

3.1.1.6 Arithmetic conversion

dblarray::dblarray(const fltarray&) throw (xnumarray)
dblarray::dblarray(const intarray&) throw (xnumarray)
fltarray::fltarray(const intarray&) throw (xnumarray)

The array created by this constructor has the same length as the argument. The values of its elements are the values of the argument elements after arithmetic conversion to the underlying type of the result array.

❑ No conversions from `dblarray` to `intarray`, from `dblarray` to `fltarray`, or from `fltarray` to `intarray` are supplied, because it would then be necessary to define mixed operations (such as `dblarray + intarray`) to avoid ambiguities. The correct way to go from `dblarray` or `fltarray` to `intarray` is through the `ceil` or `floor` functions. The correct way to go from `dblarray` to `fltarray` is through the `sngl` function. (See 3.1.1.25 on page 25.)

3.1.1.7 Destructors

dblarray::~dblarray(void)

fltarray::~fltarray(void)

intarray::~intarray(void)

❑ Nothing besides the signature need be defined by the standard. These destructors are not virtual, because experience shows that heavy use of operator overloading is incompatible with public inheritance except for the most trivial derived classes (those adding no data members). *Private* inheritance does not require a virtual destructor. The numerical array classes are best regarded as abstract data types rather than as object classes.

3.1.1.8 Assignment operators from like arrays

dblarray& dblarray::operator=(const dblarray&) throw (xnumarray)

fltarray& fltarray::operator=(const fltarray&) throw (xnumarray)

intarray& intarray::operator=(const intarray&) throw (xnumarray)

The assignment operators change the length and element values of the `this` array to be equal to those of the argument array.

❑ Assignment is the usual way to change the length of an array after initialization. It is essential that there be a way to do this, or there would be no way to create a useful array of `dblarray` or `intarray`. Also, as noted in 3.1.1.5, assignment results in conceptually distinct arrays rather than an alias.

The standard mandates a non-conforming assignment. Some programmers prefer conforming assignment, in which it is an error that may be diagnosed if the array on the left hand side of the assignment operator does not already have the same length as the array expression on the right hand side. A resizing function would be provided separately. It is claimed that a conforming assignment will detect important coding errors; however, a non-conforming assignment is likely to expose array lengths less frequently, which may prevent coding errors. The two assignments differ only slightly in their optimization characteristics and, on balance, it is largely a matter of taste which is chosen. Non-conforming assignment was chosen largely because it seems more natural for the most common implementation of the array classes (smart pointers with reference counting).

3.1.1.9 Assignment with arithmetic conversion

dblarray& dblarray::operator=(const fltarray&) throw (xnumarray)

dblarray& dblarray::operator=(const intarray&) throw (xnumarray)

fltarray& fltarray::operator=(const intarray&) throw (xnumarray)

These operators change the length of the `this` array to be equal to that of the argument array. The values of the elements of the `this` array are set equal to the values of the elements of the argument array after arithmetic conversion to the underlying type of the `this` array

❑ These operations may not seem necessary, since the conversion and assignment can take place through two operations that have already been defined. However, the direct assignment can be significantly more efficient than the two-step process, and it seemed unacceptable to prohibit a direct assignment or to make the existence of a direct assignment implementation dependent.

3.1.1.10 Length access

size_t dblarray::length(void) const

size_t fltarray::length(void) const

size_t intarray::length(void) const

These functions return the length of the `this` array.

3.1.1.11 Element access

double dblarray::operator[](size_t) const

double& dblarray::operator[](size_t)
float fltarray::operator[](size_t) const
float& fltarray::operator[](size_t)
int intarray::operator[](size_t) const
int& intarray::operator[](size_t)

When applied to a constant array, the subscript operator returns the value of the corresponding element of the array. When applied to a non-constant array, the subscript operator returns a reference to the corresponding element of the array. Thus, the expression

```
(a[i] = q, a[i]) == q
```

evaluates as true for any non-constant `dblarray a`, any `double q`, and for any `size_t i` such that the value of `i` is less than the length of `a`.

The expression

```
&a[i+j] == &a[i] + j
```

evaluates as true for all `size_t i` and `size_t j` such that `i+j` is less than the length of the non-constant array `a`. Likewise, the expression

```
&a[i] != &b[j]
```

evaluates as true for any two non-constant arrays `a` and `b` and for any `size_t i` and `size_t j` such that `i` is less than the length of `a` and `j` is less than the length of `b`.

❑ The latter property indicates an absence of aliasing and may be used to advantage by optimizing compilers.

The reference returned by the subscript operator for a non-constant array is guaranteed to be valid until the array to whose data it refers appears on the left side of an assignment statement, is passed into any library function through a non-constant reference, or is passed as `this` into any non-constant member function. Computed assigns [such as `dblarray& dblarray::operator+=(const dblarray&)`] do *not* by themselves invalidate references to array data.

❑ Thus, references to array elements have a useful lifetime. This minimum lifetime is well defined.

If the subscript operator is invoked with a `size_t` argument whose value is not less than the length of the array, the behavior is undefined.

❑ High-quality implementations will supply options (perhaps toggled by NDEBUG or a compiler flag) under which bounds violations are detected and reported. (See the rationale under 3.1. for an example.) However, it was not thought proper to mandate this in the standard.

3.1.1.12 Pointer conversion

dblarray::operator double*(void)
dblarray::operator const double*(void) const
fltarray::operator float*(void)
fltarray::operator const float*(void) const
intarray::operator int*(void)
intarray::operator const int*(void) const

A non-constant array may be converted to a pointer to the underlying type. A constant array may be converted to a pointer to the underlying type, qualified by `const`. It is guaranteed that

```
&a[0] == (double*)a
&b[0] == (float*)b
```

```
&c[0] == (int*)c
```

for any non-constant `dblarray a`, `fltarray b`, or `intarray c`. The pointer returned for a non-constant array (whether or not it points to a type qualified by `const`) is valid for the same duration as a reference returned by the subscript operator (see 3.1.1.11). The pointer returned for a constant array is valid for the lifetime of the array.

❑ This form of access is essential for reusability and cross-language programming. For example, a user might wish to pass an array into a commercial FORTRAN-77 numerical package as if it was a pointer to double, using `extern "FORTRAN"` linkage.

There is also some advantage in having a clumsy but potentially fast way to access elements of the array. By converting an array to a pointer and using the subscript operator on the pointer, the user can avoid any extra overhead associated with the array subscript operator (such as implementation-defined "copy-on-reference" checks).

3.1.1.13 Unary operators

dblarray dblarray::operator+(void) const throw(xnumarray)
dblarray dblarray::operator-(void) const throw(xnumarray)
fltarray fltarray::operator+(void) const throw(xnumarray)
fltarray fltarray::operator-(void) const throw(xnumarray)
intarray intarray::operator+(void) const throw(xnumarray)
intarray intarray::operator-(void) const throw(xnumarray)
intarray intarray::operator~(void) const throw(xnumarray)
intarray intarray::operator!(void) const throw(xnumarray)

Each of these operators performs the indicated operation and returns the result. To illustrate, the expression

```
(@a)[i] == @(a[i])
```

evaluates as true when `a` is an array, the value of `size_t i` is less than the length of `a`, and `@` is one of the operations overloaded above.

3.1.1.14 Binary arithmetic operators with scalars

dblarray operator*(const dblarray&, double) throw(xnumarray)
dblarray operator/(const dblarray&, double) throw(xnumarray)
dblarray operator+(const dblarray&, double) throw(xnumarray)
dblarray operator-(const dblarray&, double) throw(xnumarray)
dblarray operator*(double, const dblarray&) throw(xnumarray)
dblarray operator/(double, const dblarray&) throw(xnumarray)
dblarray operator+(double, const dblarray&) throw(xnumarray)
dblarray operator-(double, const dblarray&) throw(xnumarray)
fltarray operator*(const fltarray&, float) throw(xnumarray)
fltarray operator/(const fltarray&, float) throw(xnumarray)
fltarray operator+(const fltarray&, float) throw(xnumarray)
fltarray operator-(const fltarray&, float) throw(xnumarray)
fltarray operator*(float, const fltarray&) throw(xnumarray)
fltarray operator/(float, const fltarray&) throw(xnumarray)
fltarray operator+(float, const fltarray&) throw(xnumarray)
fltarray operator-(float, const fltarray&) throw(xnumarray)
intarray operator*(const intarray&, int) throw(xnumarray)
intarray operator/(const intarray&, int) throw(xnumarray)
intarray operator%(const intarray&, int) throw(xnumarray)

intarray operator+(const intarray&, int) throw(xnumarray)
intarray operator-(const intarray&, int) throw(xnumarray)
intarray operator*(int, const intarray&) throw(xnumarray)
intarray operator/(int, const intarray&) throw(xnumarray)
intarray operator%(int, const intarray&) throw(xnumarray)
intarray operator+(int, const intarray&) throw(xnumarray)
intarray operator-(int, const intarray&) throw(xnumarray)

Each of these operators performs the indicated operation and returns the result. Thus, the expressions

```
(a @ c)[i] == a[i] @ c
(c @ a)[i] == c @ a[i]
```

will evaluate as true for any array `a`, any scalar `c`, and any `size_t i` whose value is less than the length of the array `a`, assuming that `@` is one of the above operators and that the value of the expression on the right hand side of the == is defined.

❑ The operators are treated as friends in order to preserve the symmetry of the operation. In general, these operations will have no discernible side effects; thus, an optimizing compiler may perform algebraic transformations such as elimination of common subexpressions or reduction in strength.

❑ Ambiguities will likely arise if a programmer attempts to use an unqualified floating literal in `fltarray` expressions such as

```
fltarray a, b;
/* ... */
a = 3.0 * b;
```

This ambiguity arises from the fact that unqualified floating literals are regarded as having type `double`. Since an important reason for having `fltarray` is to permit users to avoid double precision arithmetic on machines where this is significantly slower than single precision, the alternative of making the scalar type `double` in operations with `fltarray` was rejected. Users will have to explicitly qualify floating literals instead:

```
fltarray a, b;
/* ... */
a = 3.0f * b;
```

3.1.1.15 Arithmetic computed assigns with scalars

dblarray& dblarray::operator*=(double)
dblarray& dblarray::operator/=(double)
dblarray& dblarray::operator+=(double)
dblarray& dblarray::operator-=(double)
fltarray& fltarray::operator*=(float)
fltarray& fltarray::operator/=(float)
fltarray& fltarray::operator+=(float)
fltarray& fltarray::operator-=(float)
intarray& intarray::operator*=(int)
intarray& intarray::operator/=(int)
intarray& intarray::operator%=(int)
intarray& intarray::operator+=(int)
intarray& intarray::operator-=(int)

Each of these operators performs the indicated operation and stores the result in the `this` array. The `this` array is then returned by reference. Thus, the expression

```
r = (a[i] @ c), (a @= c), r == a[i]
```

will evaluate as true for any array `a`, any scalars `r` and `c`, and any `size_t i` whose value is less than the length of the array `a`, assuming that `@` is one of the above operators and that the value of the expression `(a[i] @ c)` is defined.

The appearance of an array on the left hand side of a computed assignment does *not* invalidate references or pointers to the elements of the array; see 3.1.1.11 on page 16.

❑ The computed assignments correspond exactly to the arithmetic operators in the sense that

```
a @= c;
```

and

```
a = a @ c;
```

are equivalent, except that the second form may invalidate references and pointers. The second form may therefore be safely transformed into the first by an optimizing compiler, if this is a reduction in strength for the particular implementation.

3.1.1.16 Binary arithmetic operations with other arrays

dblarray operator*(const dblarray&, const dblarray&) throw(xnumarray)
dblarray operator/(const dblarray&, const dblarray&) throw(xnumarray)
dblarray operator+(const dblarray&, const dblarray&) throw(xnumarray)
dblarray operator-(const dblarray&, const dblarray&) throw(xnumarray)
fltarray operator*(const fltarray&, const fltarray&) throw(xnumarray)
fltarray operator/(const fltarray&, const fltarray&) throw(xnumarray)
fltarray operator+(const fltarray&, const fltarray&) throw(xnumarray)
fltarray operator-(const fltarray&, const fltarray&) throw(xnumarray)
intarray& operator*(const intarray&, const intarray&) throw(xnumarray)
intarray& operator/(const intarray&, const intarray&) throw(xnumarray)
intarray& operator%(const intarray&, const intarray&) throw(xnumarray)
intarray& operator+(const intarray&, const intarray&) throw(xnumarray)
intarray& operator-(const intarray&, const intarray&) throw(xnumarray)

These operators perform the indicated operation element-by-element and return the result. If the two array arguments do not have the same length, the behavior is undefined. Thus, the expression

```
(a @ b)[i] == a[i] @ b[i]
```

will evaluate as true for any array `a` and `b` and any `size_t i` whose value is less than the length of the arrays `a` and `b`, assuming that `@` is one of the above operators and that the value of the expression on the right hand side of the == is defined.

❑ The array classes are nothing more than ordered sets of `double/int`. They are intended as building blocks for matrix or field classes, but do not have any calculus or topology of their own. Thus, the only operations defined between two arrays are element-by-element operations.

As noted in the rationale for 3.1.1.11, many implementations will implement a debug mode where array sizes will be checked for conformance in binary operations. A mechanism for doing this is suggested in the rationale for 3.1.

3.1.1.17 Arithmetic computed assignments with other arrays

dblarray& dblarray::operator*=(const dblarray&)

dblarray& dblarray::operator/=(const dblarray&)
dblarray& dblarray::operator+=(const dblarray&)
dblarray& dblarray::operator-=(const dblarray&)
fltarray& fltarray::operator*=(const fltarray&)
fltarray& fltarray::operator/=(const fltarray&)
fltarray& fltarray::operator+=(const fltarray&)
fltarray& fltarray::operator-=(const fltarray&)
intarray& intarray::operator*=(const intarray&)
intarray& intarray::operator/=(const intarray&)
intarray& intarray::operator%=(const intarray&)
intarray& intarray::operator+=(const intarray&)
intarray& intarray::operator-=(const intarray&)

Each of these operators performs the indicated operation element-by-element and stores the result in the `this` array. The `this` array is then returned by reference. Thus, the expression

```
r = a[i] @ b[i], a @= b, r == a[i]
```

will evaluate as true for any array `a` and `b`, scalar `r`, and `size_t i` whose value is less than the length of the arrays `a` and `b`, assuming that `@` is one of the above operators and that the value of the expression `(a[i] @ b[i])` is defined. If the `this` array and the argument array do not have the same length, the behavior is undefined.

The appearance of an array on the left hand side of a computed assignment does *not* invalidate references or pointers.

❑ See 3.1.1.15 on page 19.

3.1.1.18 Logical and bitwise operators with scalar

intarray operator==(const dblarray&, double) throw(xnumarray)
intarray operator!=(const dblarray&, double) throw (xnumarray)
intarray operator<(const dblarray&, double) throw (xnumarray)
intarray operator>(const dblarray&, double) throw (xnumarray)
intarray operator<=(const dblarray&, double) throw (xnumarray)
intarray operator>=(const dblarray&, double) throw (xnumarray)
intarray operator==(double, const dblarray&) throw(xnumarray)
intarray operator!=(double, const dblarray&) throw (xnumarray)
intarray operator<(double, const dblarray&) throw (xnumarray)
intarray operator>(double, const dblarray&) throw (xnumarray)
intarray operator<=(double, const dblarray&) throw (xnumarray)
intarray operator>=(double, const dblarray&) throw (xnumarray)
intarray operator==(const fltarray&, float) throw(xnumarray)
intarray operator!=(const fltarray&, float) throw (xnumarray)
intarray operator<(const fltarray&, float) throw (xnumarray)
intarray operator>(const fltarray&, float) throw (xnumarray)
intarray operator<=(const fltarray&, float) throw (xnumarray)
intarray operator>=(const fltarray&, float) throw (xnumarray)
intarray operator==(float, const fltarray&) throw(xnumarray)
intarray operator!=(float, const fltarray&) throw (xnumarray)
intarray operator<(float, const fltarray&) throw (xnumarray)
intarray operator>(float, const fltarray&) throw (xnumarray)
intarray operator<=(float, const fltarray&) throw (xnumarray)

intarray operator>=(float, const fltarray&) throw (xnumarray)
intarray operator==(const intarray&, int) throw(xnumarray)
intarray operator!=(const intarray&, int) throw (xnumarray)
intarray operator<(const intarray&, int) throw (xnumarray)
intarray operator>(const intarray&, int) throw (xnumarray)
intarray operator<=(const intarray&, int) throw (xnumarray)
intarray operator>=(const intarray&, int) throw (xnumarray)
intarray operator^(const intarray&, int) throw (xnumarray)
intarray operator&(const intarray&, int) throw (xnumarray)
intarray operator|(const intarray&, int) throw (xnumarray)
intarray operator<<(const intarray&, int) throw (xnumarray)
intarray operator>>(const intarray&, int) throw (xnumarray)
intarray operator&&(const intarray&, int) throw (xnumarray)
intarray operator||(const intarray&, int) throw (xnumarray)
intarray operator==(int, const intarray&) throw(xnumarray)
intarray operator!=(int, const intarray&) throw (xnumarray)
intarray operator<(int, const intarray&) throw (xnumarray)
intarray operator>(int, const intarray&) throw (xnumarray)
intarray operator<=(int, const intarray&) throw (xnumarray)
intarray operator>=(int, const intarray&) throw (xnumarray)
intarray operator^(int, const intarray&) throw (xnumarray)
intarray operator&(int, const intarray&) throw (xnumarray)
intarray operator|(int, const intarray&) throw (xnumarray)
intarray operator<<(int, const intarray&) throw (xnumarray)
intarray operator>>(int, const intarray&) throw (xnumarray)
intarray operator&&(int, const intarray&) throw (xnumarray)
intarray operator||(int, const intarray&) throw (xnumarray)

Each of these operators performs the indicated operation and returns the result. Thus, the expressions

```
(a @ c)[i] == (a[i] @ c)
(c @ a)[i] == (c @ a[i])
```

will evaluate as true for any array `a`, any scalar c, and any `size_t i` whose value is less than the length of the array `a`, assuming that `@` is one of the above operators and that the value of the expression on the right hand side of the == is defined.

3.1.1.19 Logical and bitwise operations with other arrays

intarray operator==(const dblarray&, const dblarray&) throw(xnumarray)
intarray operator!=(const dblarray&, const dblarray&) throw (xnumarray)
intarray operator<(const dblarray&, const dblarray&) throw (xnumarray)
intarray operator>(const dblarray&, const dblarray&) throw (xnumarray)
intarray operator<=(const dblarray&, const dblarray&) throw (xnumarray)
intarray operator>=(const dblarray&, const dblarray&) throw (xnumarray)
intarray operator==(const fltarray&, const fltarray&) throw(xnumarray)
intarray operator!=(const fltarray&, const fltarray&) throw (xnumarray)
intarray operator<(const fltarray&, const fltarray&) throw (xnumarray)
intarray operator>(const fltarray&, const fltarray&) throw (xnumarray)
intarray operator<=(const fltarray&, const fltarray&) throw (xnumarray)

intarray operator>=(const fltarray&, const fltarray&) throw (xnumarray)
intarray operator==(const intarray&, const intarray&) throw(xnumarray)
intarray operator!=(const intarray&, const intarray&) throw (xnumarray)
intarray operator<(const intarray&, const intarray&) throw (xnumarray)
intarray operator>(const intarray&, const intarray&) throw (xnumarray)
intarray operator<=(const intarray&, const intarray&) throw (xnumarray)
intarray operator>=(const intarray&, const intarray&) throw (xnumarray)
intarray operator^(const intarray&, const intarray&) throw (xnumarray)
intarray operator&(const intarray&, const intarray&) throw (xnumarray)
intarray operator|(const intarray&, const intarray&) throw (xnumarray)
intarray operator<<(const intarray&, const intarray&) throw (xnumarray)
intarray operator>>(const intarray&, const intarray&) throw (xnumarray)
intarray operator&&(const intarray&, const intarray&) throw (xnumarray)
intarray operator||(const intarray&, const intarray&) throw (xnumarray)

These operators perform the indicated operation element-by-element and return the result. If the two array arguments do not have the same length, the behavior is undefined. Thus, the expression

```
(a @ b)[i] == (a[i] @ b[i])
```

will evaluate as true for any array `a` and `b` and any `size_t i` whose value is less than the length of the arrays `a` and `b`, assuming that `@` is one of the above operators and that the value of the expression on the right hand side of the == is defined.

3.1.1.20 Logical and bitwise computed assignments with scalar

intarray& intarray::operator^=(int)
intarray& intarray::operator&=(int)
intarray& intarray::operator|=(int)
intarray& intarray::operator<<=(int)
intarray& intarray::operator>>=(int)

Each of these operators performs the indicated operation element-by-element and stores the result in the `this` array. The `this` array is then returned by reference. Thus, the expression

```
r = a[i] @ c, a @= c, r == a[i]
```

will evaluate as true for any array `a,` scalars `r` and `c`, and `size_t i` whose value is less than the length of `a`, assuming that `@` is one of the above operators and that the value of the expression `(a[i] @ c)` is defined.

The appearance of an array on the left hand side of a computed assignment does *not* invalidate references or pointers.

❑ See 3.1.1.15 on page 19.

3.1.1.21 Logical and bitwise computed assignments with other arrays

intarray& intarray::operator^=(const intarray&)
intarray& intarray::operator&=(const intarray&)
intarray& intarray::operator|=(const intarray&)
intarray& intarray::operator<<=(const intarray&)
intarray& intarray::operator>>=(const intarray&)

Each of these operators performs the indicated operation element-by-element and stores the result in the `this` array. The `this` array is then returned by reference. Thus, the expression

```
r = a[i] @ b[i], a @= b, r == a[i]
```

will evaluate as true for any array a and b, scalar r, and `size_t i` whose value is less than the length of the arrays a and b, assuming that @ is one of the above operators and that the value of the expression `(a[i] @ b[i])` is defined. If the `this` array and the argument array do not have the same length, the behavior is undefined.

The appearance of an array on the left hand side of a computed assignment does *not* invalidate references or pointers.

❑ See 3.1.1.15 on page 19.

3.1.1.22 Selection functions

dblarray where(const intarray&, const dblarray&, const dblarray&) throw(xnumarray)
fltarray where(const intarray&, const fltarray&, const fltarray&) throw(xnumarray)
intarray where(const intarray&, const intarray&, const intarray&) throw(xnumarray)‘
dblarray where(const intarray&, const dblarray&, double) throw (xnumarray)
dblarray where(const intarray&, double, const dblarray&) throw (xnumarray)
dblarray where(const intarray&, double, double) throw (xnumarray)
fltarray where(const intarray&, const fltarray&, float) throw (xnumarray)
fltarray where(const intarray&, float, const fltarray&) throw (xnumarray)
fltarray where(const intarray&, float, float) throw (xnumarray)
intarray where(const intarray&, const intarray&, int) throw (xnumarray)
intarray where(const intarray&, int, const intarray&) throw (xnumarray)
intarray where(const intarray&, int, int) throw (xnumarray)

The `where` function resembles the `?:` operator. In the first form, it returns an array whose length is equal to the lengths of the three argument arrays. If the three argument arrays do not have the same length, the behavior is undefined. If an element of the first argument is nonzero, the corresponding element of the result will be equal to the corresponding element of the second argument; otherwise, it will be equal to the corresponding element of the third argument. That is, the expression

```
where(a, b, c)[i] == (a[i] ? b[i] : c[i])
```

will evaluate as true for arrays `a, b`, and `c` and for any `size_t i` whose value is less than the length of the arrays.

The mixed forms are similar in behavior; thus, the expressions

```
where(a, b, sc)[i] == (a[i] ? b[i] : sc)
where(a, sc, b)[i] == (a[i] ? sc : b[i])
where(a, sc1, sc2)[i] == (a[i] ? sc1 : sc2)
```

will all evaluate as true when a and b are arrays of identical length, sc, sc1, and sc2 are scalars, and the value of `size_t i` is less than the length of the argument arrays.

❑ The overloading of ?: is undefined by the standard. It was felt that this single useful instance of an overloaded ?: was not sufficient justification for changing existing practice, especially since the functional form is nearly as expressive.

Note that if the second and/or third parameters are array expressions without side effects, the unselected elements of each expression need not be computed. However, this optimization is almost impossible to make without built-in compiler support for the numerical array classes.

3.1.1.23 Trace functions

double dblarray::sum(void) const
float fltarray::sum(void) const
int intarray::sum(void) const

These functions return the sum of all the elements of an array.

❑ For example, if one is interpreting the array as a vector, the inner product can be defined as

```
inline InnerProduct(const dblarray& a, const dblarray& b){
  return (a*b).sum();
}
```

3.1.1.24 Fill functions

void dblarray::fill(double)

void fltarray::fill(float)

void intarray::fill(int)

These functions set the values of all the elements of the `this` array equal to the argument. The length of the array is not changed, nor are any pointers or references to the elements of the array invalidated.

❑ These functions allow the programmer to reset the values of a preexisting array without exposing its length.

Some data-parallel languages support an assignment from scalar whose semantics resemble the fill functions. In other words,

```
a.fill(c);
```

would be replaced in these languages by

```
a = c;
```

where `a` is an array and `c` is a scalar. Although some programmers desired an assignment from scalar in the `<numarray>` classes, others considered this construct confusing and dangerous, and it was ultimately rejected.

3.1.1.25 Transcendentals

dblarray acos(const dblarray&) throw (xnumarray)

dblarray asin(const dblarray&) throw (xnumarray)

dblarray atan(const dblarray&) throw (xnumarray)

dblarray atan2(const dblarray&, const dblarray&) throw (xnumarray)

dblarray atan2(const dblarray&, double) throw (xnumarray)

dblarray atan2(double, const dblarray&) throw (xnumarray)

intarray ceil(const dblarray&) throw (xnumarray)

dblarray cos(const dblarray&) throw (xnumarray)

dblarray cosh(const dblarray&) throw (xnumarray)

dblarray exp(const dblarray&) throw (xnumarray)

dblarray fabs(const dblarray&) throw (xnumarray)

intarray floor(const dblarray&) throw (xnumarray)

dblarray frexp(const dblarray&, intarray&) throw (xnumarray)

dblarray fmod(const dblarray&, const dblarray&) throw (xnumarray)

dblarray fmod(const dblarray&, double) throw (xnumarray)

dblarray fmod(double, const dblarray&) throw (xnumarray)

dblarray ldexp(const dblarray&, const intarray&) throw (xnumarray)

dblarray log(const dblarray&) throw (xnumarray)

dblarray log10(const dblarray&) throw (xnumarray)

dblarray modf(const dblarray&, dblarray&) throw (xnumarray)

dblarray pow(const dblarray&, const dblarray&) throw (xnumarray)

dblarray pow(const dblarray&, double) throw (xnumarray)

dblarray pow(double, const dblarray&) throw (xnumarray)

dblarray sin(const dblarray&) throw (xnumarray)

dblarray sinh(const dblarray&) throw (xnumarray)
dblarray sqrt(const dblarray&) throw (xnumarray)
dblarray tan(const dblarray&) throw (xnumarray)
dblarray tanh(const dblarray&) throw (xnumarray)
fltarray acos(const fltarray&) throw (xnumarray)
fltarray asin(const fltarray&) throw (xnumarray)
fltarray atan(const fltarray&) throw (xnumarray)
fltarray atan2(const fltarray&, const fltarray&) throw (xnumarray)
fltarray atan2(const fltarray&, float) throw (xnumarray)
fltarray atan2(float, const fltarray&) throw (xnumarray)
intarray ceil(const fltarray&) throw (xnumarray)
fltarray cos(const fltarray&) throw (xnumarray)
fltarray cosh(const fltarray&) throw (xnumarray)
fltarray exp(const fltarray&) throw (xnumarray)
fltarray fabs(const fltarray&) throw (xnumarray)
intarray floor(const fltarray&) throw (xnumarray)
fltarray frexp(const fltarray&, intarray&) throw (xnumarray)
fltarray fmod(const fltarray&, const fltarray&) throw (xnumarray)
fltarray fmod(const fltarray&, float) throw (xnumarray)
fltarray fmod(float, const fltarray&) throw (xnumarray)
fltarray ldexp(const fltarray&, const intarray&) throw (xnumarray)
fltarray log(const fltarray&) throw (xnumarray)
fltarray log10(const fltarray&) throw (xnumarray)
fltarray modf(const fltarray&, fltarray&) throw (xnumarray)
fltarray pow(const fltarray&, const fltarray&) throw (xnumarray)
fltarray pow(const fltarray&, float) throw (xnumarray)
fltarray pow(float, const fltarray&) throw (xnumarray)
fltarray sin(const fltarray&) throw (xnumarray)
fltarray sngl(const dblarray&) throw (xnumarray)
fltarray sinh(const fltarray&) throw (xnumarray)
fltarray sqrt(const fltarray&) throw (xnumarray)
fltarray tan(const fltarray&) throw (xnumarray)
fltarray tanh(const fltarray&) throw (xnumarray)
intarray abs(const intarray&) throw(xnumarray)
void div(const intarray&, const intarray&, intarray&, intarray&) throw(xnumarray)
void intarray::rand(void)

These functions correspond closely to the scalar functions found in `<math.h>` and `<stdlib.h>`. For example, the expressions

```
acos(a)[i] == acos(a[i])
atan2(a, b)[i] == atan2(a[i], b[i])
(modf(a, b)[i] == modf(a[i], &sc)) && (b[i] == sc)
```

will all evaluate as true for `dblarray` `a` and `b`, `double sc`, and `size_t i` whose value is less than the length of the constant array arguments, assuming the values of the expressions on the right-hand side of an == are defined. For the functions taking more than one constant array argument, if the lengths of the constant array arguments are not equal, the behavior is undefined. Domain or range errors analogous to those for the corresponding scalar functions are possible.

The `rand` function replaces the values of all the elements of the `this` array with pseudo-random values between `0` and `RAND_MAX` inclusive. The length of the `this` array remains unchanged. The pseudo-random sequence is seeded by a call to `void srand(unsigned int)`. The relationship between `void intarray::rand(void)`, `int rand(void)`, and `void srand(unsigned int)` is otherwise implementation defined.

❑ The `rand` function can supply an array of pseudo-random values for Monte Carlo calculations.

In many implementations, the array `rand` will make use of the scalar `rand` and put successive terms of the sequence returned by scalar `rand` in successive elements of the array. However, this may be overly restrictive for some high-performance architectures, such as massively parallel computers. The implementation must document how array `rand` is actually implemented.

The functions `ceil` and `floor` differ slightly from their scalar counterparts in that they return an `intarray` rather than a `dblarray` or `fltarray`. The function `sngl` has no counterpart in the scalar math library. It converts a `dblarray` to a `fltarray` of identical length whose elements have the values of the elements of the argument array after coercion to type `float`.

❑ This was done to provide a conversion from `dblarray` or `fltarray` to `intarray` and from `dblarray` to `fltarray` without defining a conversion constructor that would have led to ambiguity in mixed operations (see 3.1.1.6 on page 15).

3.1.1.26 FORTRAN-like functions

dblarray dim(const dblarray&, const dblarray&) throw (xnumarray)
dblarray dim(const dblarray&, double) throw (xnumarray)
dblarray dim(double, const dblarray&) throw (xnumarray)
dblarray min(const dblarray&, const dblarray&) throw (xnumarray)
dblarray min(const dblarray&, double) throw (xnumarray)
dblarray min(double, const dblarray&) throw (xnumarray)
double min(const dblarray&)
dblarray max(const dblarray&, const dblarray&) throw (xnumarray)
dblarray max(const dblarray&, double) throw (xnumarray)
dblarray max(double, const dblarray&) throw (xnumarray)
double max(dblarray&)
dblarray sign(const dblarray&, const dblarray&) throw (xnumarray)
dblarray sign(const dblarray&, double) throw (xnumarray)
dblarray sign(double, const dblarray&) throw (xnumarray)
fltarray dim(const fltarray&, const fltarray&) throw (xnumarray)
fltarray dim(const fltarray&, float) throw (xnumarray)
fltarray dim(float, const fltarray&) throw (xnumarray)
fltarray min(const fltarray&, const fltarray&) throw (xnumarray)
fltarray min(const fltarray&, float) throw (xnumarray)
fltarray min(float, const fltarray&) throw (xnumarray)
float min(const fltarray&)
fltarray max(const fltarray&, const fltarray&) throw (xnumarray)
fltarray max(const fltarray&, float) throw (xnumarray)
fltarray max(float, const fltarray&) throw (xnumarray)
float max(fltarray&)
fltarray sign(const fltarray&, const fltarray&) throw (xnumarray)
fltarray sign(const fltarray&, float) throw (xnumarray)
fltarray sign(float, const fltarray&) throw (xnumarray)

intarray dim(const intarray&, const intarray&) throw (xnumarray)
intarray dim(const intarray&, int) throw (xnumarray)
intarray dim(int, const intarray&) throw (xnumarray)
intarray min(const intarray&, const intarray&) throw (xnumarray)
intarray min(const intarray&, int) throw (xnumarray)
intarray min(int, const intarray&) throw (xnumarray)
int min(const intarray&)
intarray max(const intarray&, const intarray&) throw (xnumarray)
intarray max(const intarray&, int) throw (xnumarray)
intarray max(int, const intarray&) throw (xnumarray)
int max(intarray&)
intarray sign(const intarray&, const intarray&) throw (xnumarray)
intarray sign(const intarray&, int) throw (xnumarray)
intarray sign(int, const intarray&) throw (xnumarray)

Most of these functions operate on either two arrays or an array and a scalar. For the functions taking two array arguments, if the lengths of the arguments are not the same, the behavior is undefined.

The dim function returns the positive difference of its arguments; thus, the expressions

```
dim(a, b)[i] == (a[i] > b[i] ? a[i] - b[i] : 0.0)
dim(a, c)[i] == (a[i] > c ? a[i] - c : 0.0)
dim(c, b)[i] == (c > b[i] ? c - b[i] : 0.0)
```

will all evaluate as true for any arrays `a` and `b` of identical length, any scalar `c`, and any `size_t i` whose value is less than the length of the arrays.

The min function returns the minimum of its arguments; thus, the expressions

```
min(a, b)[i] == (a[i] < b[i] ? a[i] : b[i])
min(a, c)[i] == (a[i] < c ? a[i] : c)
min(c, b)[i] == (c < b[i] ? c : b[i])
```

will all evaluate as true for any arrays `a` and `b` of identical length, any scalar `c`, and any `size_t i` whose value is less than the length of the arrays.

The max function returns the maximum of its arguments; thus, the expressions

```
max(a, b)[i] == (a[i] > b[i] ? a[i] : b[i])
max(a, c)[i] == (a[i] > c ? a[i] : c)
max(c, b)[i] == (c > b[i] ? c : b[i])
```

will all evaluate as true for any arrays `a` and `b` of identical length, any scalar `c`, and any `size_t i` whose value is less than the length of the arrays.

The sign function transfers the sign of the second argument to the first and returns the result; thus, the expressions

```
sign(a, b)[i] == (b[i] < 0.0 ? -fabs(a[i]) : fabs(a[i]))
sign(a, c)[i] == (c < 0.0 ? -fabs(a[i]) : fabs(a[i]))
sign(c, b)[i] == (b[i] < 0.0 ? -fabs(c) : fabs(c))
```

will all evaluate as true for any `dblarray a` and `b` of identical length, any scalar `c`, and any `size_t i` whose value is less than the length of the arrays.

The `min` and `max` functions taking a single array argument return the minimum or maximum of all the elements of the argument.

❑ Aside from the utility of these functions, they facilitate conversion of FORTRAN codes to C++, particularly by machine translators.

3.1.1.27 Shift functions

dblarray dblarray::shift(int) const throw(xnumarray)
fltarray fltarray::shift(int) const throw(xnumarray)
intarray intarray::shift(int) const throw(xnumarray)

These functions return an array whose length is identical to the `this` array but whose element values are shifted the number of places indicated by the argument. This is best illustrated by example. The expressions

```
a.shift(-1)[i] == a[i+1];
b.shift(3)[j] == b[j-3];
```

will all evaluate as true for any arrays `a` and `b`, any `size_t i` such that `i+1` is less than the length of `a`, and any `size_t j` such that `j-3` is less than the length of `b` and `j>2`.

❑ This permits finite-difference stencils to be implemented; for example,

```
a.shift(-1) - 2.0*a + a.shift(1)
```

is the equipotential (Laplacian) stencil in one dimension, while

```
a.shift(-1) + a.shift(-N) + a.shift(1) + a.shift(N) - 4.0*a
```

is the same stencil in two dimensions (where `N` is the mesh column count).

The values of `a.shift(n)[i]` are undefined for all `n>0` and `i<n`. Likewise, the values of `a.shift(n)[i]` are undefined for all `n<0` and `i>=a.length()-abs(n).`

❑ Normally the undefined elements will immediately be overwritten by boundary condition stencils.

Some numerical programmers have expressed a preference for trimming away the undefined boundary elements and using a subset of the original array. This functionality is provided by the unity-stride gather and scatter functions (see 3.1.1.32 on page 31).

3.1.1.28 Assembly operations

dblarray& dblarray::assemble(const dblarray&, const intarray&, const intarray&)
fltarray& fltarray::assemble(const fltarray&, const fltarray&, const fltarray&)
intarray& intarray::assemble(const intarray&, const intarray&, const intarray&)

These functions add selected elements of the first argument to selected elements of the `this` array, then return a reference to the `this` array. The second and third arguments contain the indices of the selected elements, so that the expression

```
a[ia[i]] + b[ib[i]] == a.assemble(b, ia, ib)[ia[i]]
```

will evaluate as true assuming that `ia[i]` has a nonnegative value less than the length of `a` and `ib[i]` has a nonnegative value less than the length of `b` for all `size_t i` whose value is less than the length of `ia`; that `ia` contains no duplicate entries; and that the length of `ia` is the same as the length of `ib`. If these conditions are not met , the behavior is undefined. The elements of `a` whose indices are not found in `ia` remain unchanged.

dblarray& dblarray::assemble(const dblarray&, const intarray&)
fltarray& fltarray::assemble(const fltarray&, const intarray&)
intarray& intarray::assemble(const intarray&, const intarray&)

These functions are similar to the previous functions, except that *all* the elements of the first argument are added to selected elements of the `this` array. Thus, the expression

```
a[ia[i]] + b[i] == a.assemble(b, ia)[ia[i]]
```

will evaluate as true assuming that `ia[i]` has a nonnegative value less than the length of `a` for all `size_t i` whose value is less than the length of `ia`; that `ia` has the same length as `b`; and that `ia` contains no duplicate entries. If these conditions are not met, the behavior is undefined. The elements of `a` whose indices are not found in `ia` remain unchanged.

dblarray& dblarray::assemble(double, const intarray& ia)

fltarray& fltarray::assemble(float, const intarray& ia)

intarray& intarray::assemble(int, const intarray& ia)

These functions are similar to the previous functions, except that a constant value is added to selected elements of the `this` array. Thus, the expression

```
a[ia[i]] + c == a.assemble(c, ia)[ia[i]]
```

will evaluate as true assuming that `ia[i]` has a nonnegative value less than the length of `a` for all `size_t i` whose value is less than the length of `ia` and that `ia` contains no duplicate entries. If these conditions are not met, the behavior is undefined. The elements of `a` whose indices are not found in `ia` remain unchanged.

❑ The restriction on duplicate entries in the destination index array `ia` is necessary to permit the function to be vectorized on appropriate architectures. Thus, a destination index array with duplicate entries must be decomposed into color index arrays that have no duplicate entries [1]. Note that `ib` is permitted to have duplicate entries.

These functions supports, e.g., finite element methods, where contributions from individual elements must be gathered and summed into a global array. They may also be useful for sparse matrix representations using an indexed list.

3.1.1.29 Indexed scatter operations

dblarray& dblarray::scatter(const dblarray&, const intarray&, const intarray&)

fltarray& fltarray::scatter(const fltarray&, const intarray&, const intarray&)

intarray& intarray::scatter(const intarray&, const intarray&, const intarray&)

These functions replace selected elements of the `this` array with selected elements of its first argument, then return a reference to the `this` array. The second and third arguments contain the indices of the selected elements, so that the expression

```
a.scatter(b, ia, ib)[ia[i]] == b[ib[i]]
```

will evaluate as true assuming that `ia[i]` has a nonnegative value less than the length of `a` and `ib[i]` has a nonnegative value less than the length of `b` for all `size_t i` whose value is less than the length of `ia`; that `ia` contains no duplicate entries; and that the length of `ia` is the same as the length of `ib`. If these conditions are not met , the behavior is undefined. The elements of `a` whose indices are not found in `ia` remain unchanged.

dblarray& dblarray::scatter(const dblarray&, const intarray&)

fltarray& fltarray::scatter(const fltarray&, const intarray&)

intarray& intarray::scatter(const intarray&, const intarray&)

These functions are similar to the previous functions, except that *all* the elements of the first argument replace selected elements of the `this` array. Thus, the expression

```
a.scatter(b, ia)[ia[i]] == b[i]
```

will evaluate as true assuming that `ia[i]` has a nonnegative value less than the length of `a` for all `size_t i` whose value is less than the length of `ia`; that `ia` has the same length as `b`; and that `ia` contains no duplicate entries. If these conditions are not met, the behavior is undefined. The elements of `a` whose indices are not found in `ia` remain unchanged.

dblarray& dblarray::scatter(double, const intarray& ia)

fltarray& fltarray::scatter(float, const intarray& ia)

intarray& intarray::scatter(int, const intarray& ia)

These functions are similar to the previous functions, except that a constant value replaces selected elements of the `this` array. Thus, the expression

```
a.scatter(c, ia)[ia[i]] == c
```

will evaluate as true assuming that `ia[i]` has a nonnegative value less than the length of `a` for all `size_t i` whose value is less than the length of `ia` and that `ia` contains no duplicate entries. If these conditions are not met, the behavior is undefined. The elements of `a` whose indices are not found in `ia` remain unchanged.

3.1.1.30 Indexed gather operation

dblarray dblarray::gather(const intarray&) const throw (xnumarray)
fltarray fltarray::gather(const intarray&) const throw (xnumarray)
intarray intarray::gather(const intarray&) const throw (xnumarray)

These functions return a new array whose length is identical to the length of the array argument and whose element values are selected from the `this` array. To illustrate, the expression

```
a.gather(ia)[i] == a[ia[i]]
```

will evaluate as true assuming `ia[i]` has a nonnegative value less than the length of `a` for all `size_t i` less than the length of `ia`. The `intarray ia` may have a length less than, equal to, or greater than the length of `a`.

❑ This implies that the argument array may have duplicate entries.

3.1.1.31 Stride-based gather/scatter operations

dblarray dblarray::gather(size_t, size_t, size_t) const throw (xnumarray)
fltarray fltarray::gather(size_t, size_t, size_t) const throw (xnumarray)
intarray intarray::gather(size_t, size_t, size_t) const throw (xnumarray)
dblarray& dblarray::scatter(const dblarray&, size_t, size_t, size_t)
fltarray& fltarray::scatter(const dblarray&, size_t, size_t, size_t)
intarray& intarray::scatter(const dblarray&, size_t, size_t, size_t)

These functions gather or scatter elements of arrays based on a start and stop index and a stride. To illustrate, the expression

```
a.scatter(b, i, j, k)[i+n*k] == b[n]
```

will evaluate as true assuming `i+n*k` is a nonnegative integer less than or equal to `j` and that `j` is less than the length of `a`. Likewise, the expression

```
a.gather(i, j, k)[n] == a[i+n*k]
```

will evaluate as true assuming `i+n*k` is a nonnegative integer less than or equal to `j` and that `j` is less than the length of `a`.

❑ These functions are the key to implementing matrix classes. They may also be useful for finite-difference classes.

3.1.1.32 Gather/scatter operations with implied unit stride

dblarray dblarray::gather(size_t, size_t) const throw (xnumarray)
fltarray fltarray::gather(size_t, size_t) const throw (xnumarray)
intarray intarray::gather(size_t, size_t) const throw (xnumarray)
dblarray& dblarray::scatter(const dblarray&, size_t, size_t)
fltarray& fltarray::scatter(const fltarray&, size_t, size_t)
intarray& intarray::scatter(const dblarray&, size_t, size_t)

These are identical to the functions in the previous section, except that no stride is specified. Thus, the expressions

```
a.scatter(b, i, j)[i+n] == b[n]
a.gather(i, j)[n] == a[i+n]
```

will evaluate as true assuming `i+n` is a nonnegative integer less than or equal to `j` and that `j` is less than the length of `a`. If these conditions are not met, the behavior is undefined.

❑ These functions are the preferred method for computing on a subset of an array. Note that the array returned by `gather` will be distinct from the original array.

The standard specifies functions that are distinct from those in the previous section rather than specifying a default stride argument of one. This permits significantly more efficient code for the common case (unity stride) on many machines.

3.1.1.33 Stream operations

ostream& operator<<(ostream&, const volatile dblarray&) throw(xstream)
ostream& operator<<(ostream&, const volatile fltarray&) throw(xstream)
ostream& operator<<(ostream&, const volatile intarray&) throw(xstream)
istream& operator>>(istream&, dblarray&) throw (xstream, xnumarray)
istream& operator>>(istream&, fltarray&) throw (xstream, xnumarray)
istream& operator>>(istream&, intarray&) throw (xstream, xnumarray)

These functions provide efficient unformatted stream I/O for the array classes. It is guaranteed that an array written to a suitable stream using the inserter operator may subsequently be read from the same location in the stream using the extractor operator. The array that is read will have the same length and element values as the array that was written.

❑ The inserter functions take a `volatile` array argument to indicate the presence of side effects. Hence the intent of this argument qualifier is mnemonic rather than semantic. A true volatile numerical array is practically useless.

All other library functions that take only constant array reference arguments and/or a `const this` pointer (if a member function of an array class) have no discernible side effects.

3.1.1.34 Free functions

void dblarray::free(void)
void fltarray::free(void)
void intarray::free(void)

These functions set the length of an array to zero.

❑ An implementation may reclaim the storage used by the array when one of these functions is called.

3.1.2 Exceptions

As indicated by their prototypes, many array library functions are permitted to throw an `xnumarray` exception if there are not sufficient resources available to carry out the operation.

❑ Note that the exception *may* be thrown, but is not mandated. An implementation that allocates internal storage for numerical arrays from somewhere other than the heap may have a difficult time detecting the lack of resources and thus may generate exceedingly undefined behavior.

The stream functions (3.1.1.33) may throw an `xstream` exception. The form and meaning of this exception is defined in the specification for the stream library.

(This ends the formal proposal.)

4. Optimizations

The numerical array proposal has been carefully written so that the array classes will have the following properties:

1. None of the operators or functions declared in the proposal which take only constant reference or non-reference arguments or a constant `this` pointer have any discernible side effects except for the stream inserter operations. This assumes the library user has not supplied versions of the free store functions that deliberately generate side effects. Since the array classes are quite likely to make use of the free store, the presence or absence of such side effects will theoretically be discernible. However, such behavior is not defined by the Standard.

2. References and pointers to elements of arrays are free of certain forms of aliasing. See 3.1.1.11 on page 16.

3. References to numerical arrays will suffer, at worst, from the alias

   ```
   &a[i] == &b[i]
   ```

   where `a` and `b` are references to numerical arrays of the same type and `size_t i` has a value less than the length of either array. Numerical array references cannot suffer from any alias of the form

   ```
   &a[i] == &b[i+n]
   ```

   where `n` is nonzero. This guarantee is sufficient to eliminate vector recursion in most array expressions.

4. The common mathematical operators overloaded for numerical arrays may be regarded as algebraically equivalent to the corresponding operators for intrinsics. For example,

   ```
   A * X * X + B * X + C
   ```

   yields the same result as

   ```
   (A*X + B)*X + C
   ```

5. The usual correspondence between a computed-assign operator and the corresponding binary operator exists. That is,

   ```
   A = A + B;
   ```

   is equivalent to

   ```
   A += B;
   ```

   except that the former may invalidate references and pointers to data in `A`, whereas the second must not. Thus the first may be safely transformed to the second. On most implementations, this will be a significant reduction in strength.

These properties of numerical arrays permit an optimizing compiler to make correct decisions about the presence or absence of vector recursion and aliasing when optimizing array expressions. They permit algebraic transformations such as elimination of common subexpressions or reductions in strength. In general, a compiler will either have to implement the array library as built-in classes, or use optimization `#pragma` directives or propriety extensions (such as Cray's `restrict` keyword [5]) in the library header, to take advantage of these properties.

5. Implementation Experience

A library similar to the one proposed above has been implemented and is being maintained by Sandia National Laboratories in cooperation with the Department of Electrical, Electronics, and Computer Engineering at the University of New Mexico. This class has provided reasonably efficient and convenient access to the specialized hardware of Sandia's vector and massively parallel computers while permitting most code development to take place on workstations.

5.1 The Sandia/UNM Implementation

The Sandia array library classes are implemented as smart pointers to a structure containing the length of the array, a reference count, and the actual data. Storage for this structure is allocated off the heap. "Copy on reference" semantics are implemented to ensure that each field is functionally distinct. In practice, many array objects may point to a given block of data, but most of these array objects will be temporaries.

A bounds violation or attempt to operate on incompatible fields will be detected by an assertion check. This is not very useful outside of a postmortem debugger, where the call stack can be viewed. It would be very nice if an exception was thrown that printed the stack as it was unwound. However, such a suggestion is outside the scope of this paper. This bounds checking can be turned off by defining the `NDEBUG` macro, since it uses the standard C `assert` facility.

One of our codes using the array classes makes calls to FORTRAN subroutines for compute-intensive operations. The conversion from array object to pointer to double makes this possible. Since current C++ compiler technology falls behind FORTRAN from a computational efficiency standpoint, this is important for making our codes competitive. Within the library itself, most operations are implemented through calls to FORTRAN or assembler routines. This ensures maximum use of specialized hardware.

The set of topological operations (gather, scatter, stride, etc.) in the array library include those required for the two large coding groups at Sandia that use the field library. One is working on a finite-difference code and the other is working on a finite-element code. I think this pretty much spans the possible topological operations for a generic array class, but the successful use of the array library with something entirely different (image processing or a linear algebra application) would increase my confidence that all the essential topological operations are there.

The Sandia array library includes operations not included in this proposal. These operations were specific to certain applications, were redundant, or were of too little value to include in a generic class. The guiding principle was: "When in doubt, leave it out." I have also included operations in this proposal that were clearly useful and general but which we have not yet had occasion to use in our own work. For example, the Sandia library does not define a `fltarray` class.

5.2 The Sandia Public-Domain Implementation

I have built a version of the Sandia array library implementation which conforms to cfront 2.1. This prototype is highly portable (though not highly optimized) and has been placed in the public domain. To obtain a copy of the prototype, contact me at `kgbudge@sherpa.sandia.gov`; or call (505)844-8244.

5.3 The Numerical Array Library and BLAS

BLAS (Basic Linear Algebra Subprograms) is a set of linear algebra utilities that is in wide use by FORTRAN programmers today. It is fascinating to read the rationale for the library that the authors gave in 1979:

> Designers of computer programs involving linear algebraic operations have frequently chosen to implement certain low level operations such as the dot product as separate subprograms. This may be observed both in many published codes and in codes written for specific applications at many computer installations. Following are some of the reasons for taking this approach:
>
> 1. It can serve as a conceptual aid in both the design and the coding stages of a programming effort to regard an operation such as the dot product as a basic building block. This is consistent with the ideas of structured programming which encourage modularizing common code sequences.

2. It improves the self-documenting quality of code to identify an operation such as the dot product by a unique mnemonic name.

3. Since a significant amount of the execution time in complicated linear algebraic programs may be spent in a few low level operations, a reduction of the execution time spent in these operations may be reflected in cost savings in the running of programs. Assembly language coded subprograms for these operations provide such savings on some computers.

4. The programming of some of these low level operations involves algorithmic and implementation subtleties that are likely to be ignored in the typical applications programming environment. ...

   If there could be general agreement on standard names and parameter lists for some of these basic operations, it would add the additional benefit of *portability* with *efficiency* on the assumption that the assembly language subprograms were generally available. Such standard subprograms would provide building blocks with which designers of portable subprograms for higher level linear algebraic operations such as solving linear algebraic equations, eigenvalue problems, etc., could achieve additional efficiency. ...

(See [6], page 309.) These are similar to the rationale for the numerical array library, except that the numerical array library is also intended as a target for sophisticated front-end optimizations transcending the scope of individual operations.

The specific BLAS routines are slightly higher-level than the operations and functions proposed for the numerical array class, because they target a specific application (linear algebra). For example, they include a dot product function and a subroutine to evaluate the expression $y = ax + b$ for array variables. These could be coded as

```
inline double DotProduct(const dblarray& a, const dblarray& b){
  return (a*b).sum();
}
inline LinearFunction(const dblarray& a, const dblarray& b, const dblarray& x){
  return a*x + b;
}
```

in C++ using the numerical array library. An excellent test of the quality of an implementation of the numerical array library might be to use it to reproduce the BLAS routines, then compare the performance to the FORTRAN BLAS.

5.4 Other Implementations

I am aware of several array or matrix libraries by other authors, but these are all higher-level than the library in this proposal. The numerical array classes are intended to be building blocks for such higher-level classes. The authors of these classes seem to be agreed that the lack of a first-class array in C++ is a serious difficulty.

6. Conclusions

The numerical array classes described in this paper provide a way to introduce first-class numerical arrays into C++ without disturbing the basic language. On high-performance platforms, these arrays can be implemented as built-in types to take full advantage of the hardware. On other platforms, they will be an ordinary set of classes (which may take advantage of vendor extensions or `#pragma` directives) that provide users with basic, portable array functionality without imposing an undue burden on the implementor. They will provide scientists and engineers with something they now lack — namely, an efficient language for object-oriented numerics — at a relatively low cost.

7. Acknowledgment and Disclaimer

This work performed at Sandia National Laboratories supported by the U. S. Department of Energy under contract number DE-AC04-76DP00789. The opinions expressed in this paper are those of the author and do not necessarily reflect the views of the U.S. Department of Energy or the United States Government.

8. References

[1] Benson, D.J. “Vectorizing the Right-Hand Side Assembly in an Explicit Finite Element Program.” *Computer Methods in Applied Mechanics and Engineering* **73** (1989) pp. 147-152.

[2] Budge, K.G. “Optimization of Expressions Involving Array Classes.” X3J16-92-0076 / WG21-N0153, July 1992.

[3] Budge, K.G., Peery, J.S., and Robinson, A.C. “High-Performance Scientific Computing Using C++.” *Proceedings of the 1992 USENIX C++ Technical Conference*, August 1992.

[4] Ellis, M.A., and Stroustrup, B. *The Annotated C++ Reference Manual,* page 37. Addison-Wesley Publishing Company (1990): Reading, MA.

[5] Holly, M. “New Keyword for C++: Restrict.” X3J16-92-0057 / WG21-N0134, June 1992

[6] Lawson, C.L., Hanson, R.J., Kincaid, D.R., and Krogh, F.T. “Basic Linear Algebra Subprograms for Fortran Usage.” *ACM Transactions on Mathematical Software* **5** (1979), pp. 308-323.