Doc. No.: WG21/N0929=X3J16/96-0111
Date: 21 May 1996
Project: C++ Standard Library
Reply to: David Vandevoorde
vandevod@cs.rpi.edu

High-performance C++ implementations for valarrays (Rev. 2)

Description

It is not known how to implement the valarray template (as currently included in the working paper) with realistic performance using the core C++ language. By realistic performance, I think of no more than 20% increase in run-time and O(1) increase in storage requirements for typical arraywise operations.

Discussion

Todd Veldhuizen and myself have (independently) shown how template techniques can be used to implement valarray-like functionality with low overhead. Todd Veldhuizen (who coined the technique ‘‘expression templates’’) provides a detailed discussion on-line:

    http://monet.uwaterloo.ca/
           ~tveldhui/papers/Expression-Templates/exprtmpl.html

I made an implementation with most of the valarray functionality available at:

    ftp://ftp.cs.rpi.edu/pub/vandevod/Valarray/

We both report performance that is within a few percent of hand-coded C for arrays of sizes larger than about 25. For smaller arrays, the overhead can reduce performance dramatically (especially for 1 or 2 element arrays), but still much less so than alternative techniques. Indeed, while for medium-to-large sized arrays (sizes 1000 and up) expression templates lead to run-times about half those of the known alternative techniques (involving extraneous copying and/or run-time expression analysis), small arrays result in ‘‘order of magnitude’’ speed improvements when using expression templates.

There are various ways to enable expression template techniques. The one proposed here results in minimal changes for the user as to what constitutes a valid ‘‘valarray expression’’ (compared with the current specifications). It also considerably simplifies the overall specification for valarrays (no need for ‘‘helper class’’ with properties subtly different from valarrays etc.); the document would be shorter, yet not drastically different from the existing text.

Proposed Resolution

Replace the valarray specification by those below. This is mostly an adaptation of the existing (post-"Santa Cruz") text for the technique described above, but also implements a few other changes presented as (fairly minor) valarray issues; among them:

- no more gslices and gslice_arrays
- generic apply and reduce functions
- signed index/stride types (size_t becomes ptrdiff_t)
- length() becomes size()

---8<----------

26.1 Numeric type requirements

[Verse 2 mentions valarray<T>; it should probably be replaced by
  valarray<T, M> if these changes are included. Perhaps some
  adjustements in verses 1 and 3 would be desirable as well, but it
  doesn’t seem essential.]

26.3 Numeric arrays [lib.numarray]

[Footnote:
[In what follow, the types denoted by Rn and Wn (e.g., ‘R31’) are
[implementation dependent.

Header <valarray> synopsis

```
#include <cstddef>    // for ptrdiff_t
namespace std {
  template<class T, class M = void> class valarray; // An array of type T
  class slice;                                      // A BLAS-like slice
                                                    // out of an array

  // Unary operators
  template<class T, class M> valarray<T, R6>
    operator+(const valarray<T, M>&);
  template<class T, class M> valarray<T, R7>
    operator-(const valarray<T, M>&);
  template<class T, class M> valarray<T, R8>
    operator~{}(const valarray<T, M>&);
  template<class T, class M> valarray<bool, R9>
    operator!(const valarray<T, M>&);


  // Binary operators
  template<class T, class M1, class M2> valarray<T, R10>
    operator*(const valarray<T, M1>&, const valarray<T, M2>&);
  template<class T, class M> valarray<T, R11>
    operator*(const valarray<T, M>&, const T&);
  template<class T, class M> valarray<T, R12>
    operator*(const T&, const valarray<T, M>&);

  template<class T, class M1, class M2> valarray<T, R13>
    operator/(const valarray<T, M1>&, const valarray<T, M2>&);
  template<class T, class M> valarray<T, R14>
    operator/(const valarray<T, M>&, const T&);
  template<class T, class M> valarray<T, R15>
    operator/(const T&, const valarray<T, M>&);

  template<class T, class M1, class M2> valarray<T, R16>
    operator%(const valarray<T, M1>&, const valarray<T, M2>&);
  template<class T, class M> valarray<T, R17>
    operator%(const valarray<T, M>&, const T&);
  template<class T, class M> valarray<T, R18>
    operator%(const T&, const valarray<T, M>&);

  template<class T, class M1, class M2> valarray<T, R19>
    operator+(const valarray<T, M1>&, const valarray<T, M2>&);
  template<class T, class M> valarray<T, R20>
    operator+(const valarray<T, M>&, const T&);
  template<class T, class M> valarray<T, R21>
    operator+(const T&, const valarray<T, M>&);

  template<class T, class M1, class M2> valarray<T, R22>
    operator-(const valarray<T, M1>&, const valarray<T, M2>&);
  template<class T, class M> valarray<T, R23>
    operator-(const valarray<T, M>&, const T&);
  template<class T, class M> valarray<T, R24>
```

```
  operator-(const T&, const valarray<T, M>&);

template<class T, class M1, class M2> valarray<T, R25>
  operator^(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<T, R26>
  operator^(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R27>
  operator^(const T&, const valarray<T, M>&);

template<class T, class M1, class M2> valarray<T, R28>
  operator&(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<T, R29>
  operator&(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R30>
  operator&(const T&, const valarray<T, M>&);

template<class T, class M1, class M2> valarray<T, R31>
  operator|(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<T, R32>
  operator|(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R33>
  operator|(const T&, const valarray<T, M>&);

template<class T, class M1, class M2> valarray<T, R34>
  operator<<(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<T, R35>
  operator<<(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R36>
  operator<<(const T&, const valarray<T, M>&);

template<class T, class M1, class M2> valarray<T, R37>
  operator>>(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<T, R38>
  operator>>(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R39>
  operator>>(const T&, const valarray<T, M>&);

template<class T, class M1, class M2> valarray<T, R40>
  operator&&(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<T, R41>
  operator&&(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R42>
  operator&&(const T&, const valarray<T, M>&);

template<class T, class M1, class M2> valarray<T, R43>
  operator||(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<T, R44>
  operator||(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R45>
  operator||(const T&, const valarray<T, M>&);

template<class T, class M1, class M2> valarray<bool, R46>
  operator==(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<bool, R47>
  operator==(const valarray<T, M>&, const T&);
template<class T, class M> valarray<bool, R48>
  operator==(const T&, const valarray<T, M>&);

template<class T, class M1, class M2> valarray<bool, R49>
  operator!=(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<bool, R50>
  operator!=(const valarray<T, M>&, const T&);
template<class T, class M> valarray<bool, R51>
  operator!=(const T&, const valarray<T, M>&);

template<class T, class M1, class M2> valarray<bool, R52>
```

```
  operator<(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<bool, R53>
  operator<(const valarray<T, M>&, const T&);
template<class T, class M> valarray<bool, R54>
  operator<(const T&, const valarray<T, M>&);

template<class T, class M1, class M2> valarray<bool, R55>
  operator>(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<bool, R56>
  operator>(const valarray<T, M>&, const T&);
template<class T, class M> valarray<bool, R57>
  operator>(const T&, const valarray<T, M>&);

template<class T, class M1, class M2> valarray<bool, R58>
  operator<=(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<bool, R59>
  operator<=(const valarray<T, M>&, const T&);
template<class T, class M> valarray<bool, R60>
  operator<=(const T&, const valarray<T, M>&);

template<class T, class M1, class M2> valarray<bool, R61>
  operator>=(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<bool, R62>
  operator>=(const valarray<T, M>&, const T&);
template<class T, class M> valarray<bool, R63>
  operator>=(const T&, const valarray<T, M>&);

// Reductions
template<class T, class M, class Fr>
  T reduce(Fr&, const valarray<T, M>&);
template<class T, class M>
  T min(const valarray<T, M>&);
template<class T, class M>
  T max(const valarray<T, M>&);
template<class T, class M>
  T sum(const valarray<T, M>&);

// Unary functions
template<class T, class M, class Fu>
  valarray<T, R64> apply(Fu&, const valarray<T, M>&);
template<class T, class M>
  valarray<T, R65> abs(const valarray<T, M>&);
template<class T, class M>
  valarray<T, R66> acos(const valarray<T, M>&);
template<class T, class M>
  valarray<T, R67> asin(const valarray<T, M>&);
template<class T, class M>
  valarray<T, R68> atan(const valarray<T, M>&);
template<class T, class M>
  valarray<T, R69> cos(const valarray<T, M>&);
template<class T, class M>
  valarray<T, R70> cosh(const valarray<T, M>&);
template<class T, class M>
  valarray<T, R71> exp(const valarray<T, M>&);
template<class T, class M>
  valarray<T, R72> log(const valarray<T, M>&);
template<class T, class M>
  valarray<T, R73> log10(const valarray<T, M>&);
template<class T, class M>
  valarray<T, R74> sin(const valarray<T, M>&);
template<class T, class M>
  valarray<T, R75> sinh(const valarray<T, M>&);
template<class T, class M>
  valarray<T, R76> sqrt(const valarray<T, M>&);
template<class T, class M>
  valarray<T, R77> tan(const valarray<T, M>&);
```

```
  template<class T, class M>
    valarray<T, R78> tanh(const valarray<T, M>&);

  // Binary functions
  template<class T, class M1, class M2, class Fb> valarray<T, R79>
    apply(Fb&, const valarray<T, M1>&, const valarray<T, M2>&);
  template<class T, class M, class Fb> valarray<T, R80>
    apply(Fb&, const valarray<T, M>&, const T&);
  template<class T, class M, class Fb> valarray<T, R81>
    apply(Fb&, const T&, const valarray<T, M>&);

  template<class T, class M1, class M2> valarray<T, R82>
    atan2(const valarray<T, M1>&, const valarray<T, M2>&);
  template<class T, class M> valarray<T, R83>
    atan2(const valarray<T, M>&, const T&);
  template<class T, class M> valarray<T, R84>
    atan2(const T&, const valarray<T, M>&);

  template<class T, class M1, class M2> valarray<T, R85>
    min(const valarray<T, M1>&, const valarray<T, M2>&);
  template<class T, class M> valarray<T, R86>
    min(const valarray<T, M>&, const T&);
  template<class T, class M> valarray<T, R87>
    min(const T&, const valarray<T, M>&);

  template<class T, class M1, class M2> valarray<T, R88>
    max(const valarray<T, M1>&, const valarray<T, M2>&);
  template<class T, class M> valarray<T, R89>
    max(const valarray<T, M>&, const T&);
  template<class T, class M> valarray<T, R90>
    max(const T&, const valarray<T, M>&);

  template<class T, class M1, class M2> valarray<T, R91>
    pow(const valarray<T, M1>&, const valarray<T, M2>&);
  template<class T, class M> valarray<T, R92>
    pow(const valarray<T, M>&, const T&);
  template<class T, class M> valarray<T, R93>
    pow(const T&, const valarray<T, M>&);
} // End namespace std
```

1 The header <valarray> defines a class template valarray and a set of related function signatures for representing and manipulating arrays of values.

2 In ‘valarray<T, M>’, T is called the element type and M is called the storage model. The element type can be any type that satisfies the numeric type requirements of _lib.numeric.requirements_, but users can only directly instantiate valarray with storage model M=void.

[Footnote:
[The storage model allows implementations to treat functionalities
[with value and with reference semantics to be treated unformly. It
[also permits implementations based on expression templates, without
[imposing significant additional constraints on other techniques.

3 The implementation is however free to generate return values with other storage models denoted by Rn and Wn in subclause _lib.numarray_. The storage model void and the models introduced by the implementation as substitution for one or more Wn will be called ‘‘writable’’. Storage models introduced by an implementation only as subsitution for one or more Rn will by called ‘‘read-only’’.

4 Storage models denoted by Rn and Wn in subclause _lib.numarray_ may depend on the template parameters of the function templates in which they occur. However, they shall add no more than two levels of

nesting compared to the deepest nesting level of the types of the argument and/or *this.

[Footnote:
[With the implementation-dependent limit on recursive template
[instantiations, this guarantees a minimum allowable complexity for
[expressions involving valarrays.

5 The valarray classes are defined to be free of certain forms of aliasing, thus allowing operations on these classes to be optimized more aggressively.

6 These library functions are permitted to throw a bad_alloc (_lib.bad.alloc_) exception if there are not sufficient resources available to carry out the operation (except where otherwise specified). Note that the exception is not mandated.

7 An implementation is permitted to qualify any of the functions and member function declared in <valarray> as inline.

Template class valarray [lib.template.valarray]

```
namespace std {
  template<class T, class M> class valarray {
  public:
    // _lib.valarray.cons_ construct/destroy:
    valarray();
    explicit valarray(ptrdiff_t);
    valarray(ptrdiff_t, const T&);
    valarray(ptrdiff_t, const T*);
    template<class M2> valarray(const valarray<T, M2>&);
    ~valarray();

    // _lib.valarray.assign_ assignment:
    template<class M2> valarray<T, M>& operator=(const valarray<T, M2>&);
    valarray<T>& operator=(const T&);

    // _lib.valarray.access_ element access:
    T                operator[](ptrdiff_t) const;
    T&               operator[](ptrdiff_t);

    // _lib.valarray.subset_ subset operations:
    valarray<T, R1>   operator[](slice) const;
    valarray<T, W1>   operator[](slice);
    template<class M2>
      valarray<T, R2> operator[](const valarray<ptrdiff_t, M2>&) const;
    template<class M2>
      valarray<T, W2> operator[](const valarray<ptrdiff_t, M2>&);

    // _lib.valarray.mask_ assignment mask:
    template<class M2> valarray<T, W3> mask(const valarray<bool, M2>&);

    // _lib.valarray.cassign_ computed assignment:
    template<class M2> valarray<T, M>& operator*=(const valarray<T, M2>&);
    template<class M2> valarray<T, M>& operator/=(const valarray<T, M2>&);
    template<class M2> valarray<T, M>& operator%=(const valarray<T, M2>&);
    template<class M2> valarray<T, M>& operator+=(const valarray<T, M2>&);
    template<class M2> valarray<T, M>& operator-=(const valarray<T, M2>&);
    template<class M2> valarray<T, M>& operator^=(const valarray<T, M2>&);
    template<class M2> valarray<T, M>& operator|=(const valarray<T, M2>&);
    template<class M2> valarray<T, M>& operator&=(const valarray<T, M2>&);
    template<class M2> valarray<T, M>& operator<<=(const valarray<T, M2>&);
    template<class M2> valarray<T, M>& operator>>=(const valarray<T, M2>&);

    valarray<T, M>& operator*= (const T&);
```

```
    valarray<T, M>& operator/= (const T&);
    valarray<T, M>& operator%= (const T&);
    valarray<T, M>& operator+= (const T&);
    valarray<T, M>& operator-= (const T&);
    valarray<T, M>& operator^= (const T&);
    valarray<T, M>& operator&= (const T&);
    valarray<T, M>& operator|= (const T&);
    valarray<T, M>& operator<<=(const T&);
    valarray<T, M>& operator>>=(const T&);

    // _lib.valarray.members_ member functions:
    ptrdiff_t size() const;
    void resize(ptrdiff_t, const T&)
    void free();

    operator T*();
    operator const T*() const;

    valarray<T, R3> shift (int) const;
    valarray<T, R4> cshift(int) const;
  };
}
```

1 The template class valarray<T, M> is a one-dimensional smart array, with elements numbered sequentially from zero. It is a representation of the mathematical concept of an ordered set of values.

2 The intent is to specify an array template that has the minimum functionality necessary to address aliasing ambiguities and the proliferation of temporaries. Thus, the valarray template is neither a matrix class nor a field class. However, it is a very useful building block for designing such classes.

valarray constructors [lib.valarray.cons]

1 These constructors are only mandated for the standard storage model void. If an implementation introduces other storage models, the corresponding valarray types do not have to include these constructors and may provide others instead.

2 These constructors may allocate memory using the global new and new[] operators, using a new or new[] operator overloaded for the element type or using any other method. The selected method is implementation dependent.

```
valarray();
```

3 Constructs an object of class valarray<T, M>, which has zero length until the member function resize is invoked for it. This default constructor is essential, since arrays of valarray 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 resize member function.

[Footnote:
[For convenience, such objects are referred to as ‘‘arrays’’ throughout the
[remainder of subclause _lib.numarray_.

```
explicit valarray(ptrdiff_t);
```

4 The array created by this constructor has a length equal to the value of the argument. The elements of the array are constructed using the default constructor for the instantiating type T.

```
valarray(ptrdiff_t n, const T&);
```

5 The array created by this constructor has a length equal to the first argument. The elements of the array are initialized with the value of the second argument.

```
valarray(ptrdiff_t n, const T*);
```

6 The array created by this constructor has a length equal to the first argument. The values of the elements of the array are initialized with the first n values pointed to by the first argument. If the value of the second argument is greater than the number of values pointed to by the first argument, the behavior is undefined. This constructor is the preferred method for converting a C array to a valarray object.

```
template<class M2> valarray(const valarray<T, M2>&);
```

7 The array created by this constructor has the same length as the argument array. The elements are initialized with the values of the corresponding elements of the argument array. This copy constructor creates a distinct array rather than an alias. Implementations in which arrays share storage are permitted, but they must implement a copy-on-reference mechanism to ensure that arrays are conceptually distinct.

8 An implementation must support void and any other storage model it introduces as substitutions for M2.

```
~valarray();
```

9 The destructor may not throw an exception.

valarray assignment [lib.valarray.assign]

1 An implementation is not required to support assignment to valarrays with a read-only storage model.

```
template<class M2> valarray<T, M>& operator=(const valarray<T, M2>&);
```

2 Each element of the *this array is assigned the value of the corresponding element of the argument array, unless that element is masked off as the result of a call to the member function mask. The resulting behavior is undefined if the length of the argument array is not equal to the length of the *this array.

3 If the value of an element in the left hand side of a valarray assignment operator depends on the value of another element in that left hand side, the resulting behavior is undefined.

4 An implementation must support void and any other storage model it introduces as substitutions for M2.

```
valarray<T, M>& operator=(const T&);
```

5 The scalar assignment operator causes each element of the *this array to be assigned the value of the argument, unless that element is masked off as the result of a call to the member function mask.

valarray element access lib.valarray.access]

```
T  operator[](ptrdiff_t) const;
T& operator[](ptrdiff_t);
```

1 When applied to a constant array, the subscript operator returns the value of the corresponding element of the array. Valarrays with a

read-only storage model are not required to support the non-const subscript operator. When applied to a non-constant array, the subscript operator returns a reference to the corresponding element of the array.

2 Thus, the expression (a[i] = q, a[i]) == q evaluates as true for any non-constant valarray<T, W> a, any T q, and for any ptrdiff_t i such that the value of i is less than the length of a (W is a writable storage model).

3 The expression &a[i+j] == &a[i] + j evaluates as true for all ptrdiff_t i and ptrdiff_t j such that i+j is less than the length of the non-constant valarray<T, W> a, where W is a writable storage model.

4 Likewise, the expression &a[i] != &b[j] evaluates as true for any two non-constant arrays a and b (whose storage model is a writabl storage model) and for any ptrdiff_t i and ptrdiff_t j such that i is less than the length of a and j is less than the length of b. This property indicates an absence of aliasing and may be used to advantage by optimizing compilers.

[Footnote:
[Compilers may take advantage of inlining, constant propagation, loop fusion, [tracking of pointers obtained from operator new, and other techniques to [generate efficient valarrays.

5 The reference returned by the subscript operator for a non-constant array is guaranteed to be valid until the the member function resize(ptrdiff_t, const T&) (_lib.valarray.members_) is called for that array or until the lifetime of that array ends, whichever happens first.

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

valarray subset operations [lib.valarray.sub]

1 Each of these operations returns a subset of the array. Arrays whose storage model is read-only are not required to support the non-const versions of these operations.

2 The const-qualified versions return this subset as a valarray with value semantics wrt. the *this array. The non-const versions return a class template object which has reference semantics to the original array.

```
valarray<T, R1>   operator[](slice s) const;
valarray<T, W1>   operator[](slice s);
```

3 The size of the array returned is s.size(). Element numbered i of the array returned corresponds to element numbered (s.start()+i*s.stride()) of the *this array.

```
template<class M2>
  valarray<T, R2> operator[](const valarray<ptrdiff_t, M2>& a) const;
template<class M2>
  valarray<T, W2> operator[](const valarray<ptrdiff_t, M2>& a);
```

4 The size of the array returned is s.size(). Element numbered i of the array returned corresponds to element number a[i] of the *this array.

5 An implementation must support void and any other storage model it introduces as substitutions for M2.

6 [Example:

```
      valarray<double> a(1000), b(1000); // Default storage model (void)
      valarray<ptrdiff_t> p(200);
```

```
    // ...
    a[p] += (2.0*b)[slice(3, 200, 2)]; // Assign 3rd, 5th, ..., 401st
                                       // element of 2.0*b to a[p[3]],
                                       // a[p[5]], ..., a[p[401]].
    (b[slice(3, 200, 2)])[16] = 1.0;   // Equivalent to b[35] = 1.0.
--- end example ]
```

valarray assignment masking [lib.valarray.mask]

```
template<class M2> valarray<T, W3> mask(const valarray<bool, M2>&);
```

1 An implementation is not required to provide this member for read-only arrays.

2 The returned array has reference semantics wrt. the *this array and has size equal to that of the *this array. If x is a writable valarray, then &(mask(x))[i] == &x[i] for any nonnegative i smaller than x.size().

3 The returned array modifies the behavior of assignment and computed assignments in such a way that elements at positions for which the corresponding element in the argument array is false, are not modified by these operations.

4 If the array and the argument array do not have the same length, the behavior is undefined.

valarray computed assignment [lib.valarray.cassign]

1 An implementation is not required to support computed assignment for read-only storage models.

```
template<class M2> valarray<T, M>& operator*=(const valarray<T, M2>&);
template<class M2> valarray<T, M>& operator/=(const valarray<T, M2>&);
template<class M2> valarray<T, M>& operator%=(const valarray<T, M2>&);
template<class M2> valarray<T, M>& operator+=(const valarray<T, M2>&);
template<class M2> valarray<T, M>& operator-=(const valarray<T, M2>&);
template<class M2> valarray<T, M>& operator^=(const valarray<T, M2>&);
template<class M2> valarray<T, M>& operator|=(const valarray<T, M2>&);
template<class M2> valarray<T, M>& operator&=(const valarray<T, M2>&);
template<class M2> valarray<T, M>& operator<<=(const valarray<T, M2>&);
template<class M2> valarray<T, M>& operator>>=(const valarray<T, M2>&);
```

2 Each of these operators may only be instantiated for a type T to which the indicated operator can be applied. Each of these operators performs the indicated operation on each of its elements and the corresponding element of the argument array, unless the element of the *this array is masked off as the result of a call to the member function mask

3 The array is then returned by reference.

4 If the 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.

5 If the value of an element in the left hand side of a valarray computed assignment operator depends on the value of another element in that left hand side, the resulting behavior is undefined.

6 An implementation must support void and any other storage model it introduces as substitutions for M2.

```
valarray<T, M>& operator*= (const T&);
valarray<T, M>& operator/= (const T&);
```

```
valarray<T, M>& operator%= (const T&);
valarray<T, M>& operator+= (const T&);
valarray<T, M>& operator-= (const T&);
valarray<T, M>& operator^= (const T&);
valarray<T, M>& operator&= (const T&);
valarray<T, M>& operator|= (const T&);
valarray<T, M>& operator<<=(const T&);
valarray<T, M>& operator>>=(const T&);
```

7 Each of these operators may only be instantiated for a type T to which the indicated operator can be applied. Each of these operators applies the indicated operation to each element of the array and the non-array argument, unless that element of the array is masked off as the result of a call to the member function mask.

8 The array is then returned by reference.

9 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.

valarray member functions [lib.valarray.members]

```
ptrdiff_t size() const;
```

1 This function returns the number of elements in the array.

```
operator T*();
operator const T*() const;
```

2 An implementation is not required to support these operators for storage models other than void.

3 A non-constant array may be converted to a pointer to the instantiating type. A constant array may be converted to a pointer to the instantiating type, qualified by const .

4 It is guaranteed that &a[0] == (T*)a for any non-constant valarray<T, void> a. 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 ptrdiff_t subscript operator.

[Footnote:
[This form of access is essential for reusability and cross-language
[programming.

[Footnote^2:
[The same access can be acquired via &(a[0]).

```
valarray<T, R3> shift(int) const;
```

5 This function returns an array whose length is identical to the array, but whose element values are shifted the number of places indicated by the argument.

6 A positive argument value results in a left shift, a negative value in a right shift, and a zero value in no shift.

7 [Example:
If the argument has the value -2, the first two elements of the result will be constructed using the default constructor; the third element of the result will be assigned the value of the first element of the argument; etc.
--- end example]

```
valarray<T, R4> cshift(int) const;
```

8 This function returns an array whose length is identical to the array, but whose element values are shifted in a circular fashion the number of places indicated by the argument.

9 A positive argument value results in a left shift, a negative value in a right shift, and a zero value in no shift.

```
void free();
```

10 This function sets the length of an array to zero.

[Footnote:
[An implementation may reclaim the storage used by the array
[when this function is called.

```
void resize(ptrdiff_t sz, const T& c = T());
```

11 This member function changes the length of the *this array to sz and then assigns to each element the value of the second argument. Resizing invalidates all pointers and references to elements in the array.

valarray non-member operations [lib.valarray.nonmembers]

valarray unary operators [lib.valarray.unary]

```
template<class T, class M> valarray<T, R6> operator+(const valarray<T, M>&);
template<class T, class M> valarray<T, R7> operator-(const valarray<T, M>&);
template<class T, class M> valarray<T, R8> operator~(const valarray<T, M>&);
template<class T, class M> valarray<T, R9> operator!(const valarray<T, M>&);
```

1 Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type T or which may be unambiguously converted to type T.

2 Each of these operators returns an array whose length is equal to the length of the array. The value of each element of the returned array is equal to the value obtained by applying the indicated operator to the corresponding element of the argument array.

3 An implementation must support void and any other storage model it introduces as substitutions for M.

valarray binary operators [lib.valarray.binary]

```
template<class T, class M1, class M2> valarray<T, R10>
  operator*(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<T, R13>
  operator/(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<T, R16>
  operator%(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<T, R19>
  operator+(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<T, R22>
  operator-(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<T, R25>
  operator^(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<T, R28>
  operator&(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<T, R31>
  operator|(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<T, R34>
  operator<<(const valarray<T, M1>&, const valarray<T, M2>&);
```

```
template<class T, class M1, class M2> valarray<T, R37>
  operator>>(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<T, R40>
  operator&&(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<T, R43>
  operator||(const valarray<T, M1>&, const valarray<T, M2>&);
```

1 Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type T or which may be unambiguously converted to type T.

2 Each of these operators returns an array whose length is equal to the length of the arrays. The value of each element of the returned array is equal to the value obtained by applying the indicated operator to the corresponding elements of the argument arrays.

3 If the argument arrays do not have the same length, the behavior is undefined.

4 An implementation must support void and any other storage model it introduces as substitutions for M1 and M2.

```
template<class T, class M> valarray<T, R11>
  operator*(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R12>
  operator*(const T&, const valarray<T, M>&);
template<class T, class M> valarray<T, R14>
  operator/(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R15>
  operator/(const T&, const valarray<T, M>&);
template<class T, class M> valarray<T, R17>
  operator%(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R18>
  operator%(const T&, const valarray<T, M>&);
template<class T, class M> valarray<T, R20>
  operator+(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R21>
  operator+(const T&, const valarray<T, M>&);
template<class T, class M> valarray<T, R23>
  operator-(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R24>
  operator-(const T&, const valarray<T, M>&);
template<class T, class M> valarray<T, R26>
  operator^(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R27>
  operator^(const T&, const valarray<T, M>&);
template<class T, class M> valarray<T, R29>
  operator&(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R30>
  operator&(const T&, const valarray<T, M>&);
template<class T, class M> valarray<T, R32>
  operator|(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R33>
  operator|(const T&, const valarray<T, M>&);
template<class T, class M> valarray<T, R35>
  operator<<(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R36>
  operator<<(const T&, const valarray<T, M>&);
template<class T, class M> valarray<T, R38>
  operator>>(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R39>
  operator>>(const T&, const valarray<T, M>&);
template<class T, class M> valarray<T, R41>
  operator&&(const valarray<T, M>&, const T&);
```

```
template<class T, class M> valarray<T, R42>
  operator&&(const T&, const valarray<T, M>&);
template<class T, class M> valarray<T, R44>
  operator||(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R45>
  operator||(const T&, const valarray<T, M>&);
```

5 Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type T or which may be unambiguously converted to type T.

2 Each of these operators returns an array whose length is equal to the length of the array argument. The value of each element of the returned array is equal to the value obtained by applying the indicated operator to the corresponding elements of the array argument and the scalar argument.

3 An implementation must support void and any other storage model it introduces as substitutions for M.

valarray comparison operators [lib.valarray.comparison]

```
template<class T, class M1, class M2> valarray<bool, R46>
  operator==(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<bool, R49>
  operator!=(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<bool, R52>
  operator<(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<bool, R52>
  operator<(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<bool, R55>
  operator>(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<bool, R58>
  operator<=(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M1, class M2> valarray<bool, R61>
  operator>=(const valarray<T, M1>&, const valarray<T, M2>&);
```

1 Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type bool or which may be unambiguously converted to type bool.

2 Each of these operators returns an array whose length is equal to the length of the arrays. The value of each element of the returned array is equal to the value obtained by applying the indicated operator to the corresponding elements of the argument arrays.

3 If the argument arrays do not have the same length, the behavior is undefined.

4 An implementation must support void and any other storage model it introduces as substitutions for M1 and M2.

```
template<class T, class M> valarray<bool, R47>
  operator==(const valarray<T, M>&, const T&);
template<class T, class M> valarray<bool, R48>
  operator==(const T&, const valarray<T, M>&);
template<class T, class M> valarray<bool, R50>
  operator!=(const valarray<T, M>&, const T&);
template<class T, class M> valarray<bool, R51>
  operator!=(const T&, const valarray<T, M>&);
template<class T, class M> valarray<bool, R53>
  operator<(const valarray<T, M>&, const T&);
```

```
template<class T, class M> valarray<bool, R54>
  operator<(const T&, const valarray<T, M>&);
template<class T, class M> valarray<bool, R56>
  operator>(const valarray<T, M>&, const T&);
template<class T, class M> valarray<bool, R57>
  operator>(const T&, const valarray<T, M>&);
template<class T, class M> valarray<bool, R59>
  operator<=(const valarray<T, M>&, const T&);
template<class T, class M> valarray<bool, R60>
  operator<=(const T&, const valarray<T, M>&);
template<class T, class M> valarray<bool, R62>
  operator>=(const valarray<T, M>&, const T&);
template<class T, class M> valarray<bool, R63>
  operator>=(const T&, const valarray<T, M>&);
```

5 Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type bool or which may be unambiguously converted to type bool.

2 Each of these operators returns an array whose length is equal to the length of the array argument. The value of each element of the returned array is equal to the value obtained by applying the indicated operator to the corresponding elements of the array argument and the scalar argument.

4 An implementation must support void and any other storage model it introduces as substitutions for M.

valarray generic applicators and reductors [lib.valarray.generic]

1 An implementation must support void and any other storage model it introduces as substitutions for M, M1 and M2 in _lib.valarray.generic_.

```
template<class T, class M, class Fr> T reduce(Fr&, const valarray<T, M>&);
```

2 This function may only be instantiated for types Fr and T such that f(x, y) is a valid function call taking arguments of type T or const T and returning a value which is of type T or which may be unambiguously converted to type T.

3 Let f(x, y) be denoted by x @ y, and a.size() == N. Then:
reduce(a, f) == a[0] @ a[1] @ a[2] @ ... @ a[N-1]
where the grouping is unspecified (i.e., this could be evaluated left-to-right, right-to-left or by adding any valid set of parentheses). This operation will call f exacly N-1 times.

```
template<class T, class M, class Fu> valarray<T, R64>
  apply(Fu& f, const valarray<T, M>& a);
```

4 This function may only be instantiated for types Fu and T such that f(x) is a valid function call taking an argument of type T or const T and returning a value which is of type T or which may be unambiguously converted to type T.

5 The array returned is such that its element numbered i equals f(a[i]). Its length is a.size().

```
template<class T, class M1, class M2, class Fu> valarray<T, R79>
  apply(Fb& f, const valarray<T, M1>& a, const valarray<T, M1>& b);
```

6 The array returned is such that its element numbered i equals f(a[i], b[i]). Its length is a.size(). If the argument arrays do not have the same length, the behavior is undefined.

```
template<class T, class M, class Fb> valarray<T, R80>
  apply(Fb&, const valarray<T, M>& a, const T& b);
template<class T, class M, class Fb> valarray<T, R81>
  apply(Fb&, const T& a, const valarray<T, M>& b);
```

7 Each of these binary apply functions function may only be instantiated for types Fb and T such that f(x, y) is a valid function call taking arguments of type T or const T and returning a value which is of type T or which may be unambiguously converted to type T.

8 The arrays returned are such that their element numbered i equals f(a[i], b) and f(a, b[i]) respectively. Their length is equal to the length of the argument array.

9 It is unspecified how many times f will be called for any of these (unary and binary) apply functions.

valarray reductions [lib.valarray.reductions]

1 An implementation must support void and any other storage model it introduces as substitutions for M in the reduction templates.

```
template<class T, class M> T min(const valarray<T, M>& a);
template<class T, class M> T max(const valarray<T, M>& a);
```

2 These functions may only be instantiated for a type T to which operator> and operator< may be applied and for which operator> and operator< return a value which is of type \bool or a type which can be unambiguously converted to type bool.

3 These functions return the minimum (min(a)) and maximum (max(a)) value found in the argument array a.

4 The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using operator> and operator<, in a manner analogous to the application of f in reduce(f, a).

```
template<class T, class M> T sum(const valarray<T, M>&);
```

5 This function may only be instantiated for a type T to which operator+= can be applied. This function returns the sum of all the elements of the argument array.

6 If the array has length 0, the behavior is undefined. If it has length 1, sum returns the value of element 0. Otherwise, the returned value is calculated by applying operator+= in a manner analoguous to the application of f in reduce(f, a).

valarray unary functions [lib.valarray.ufuncs]

```
template<class T, class M> valarray<T, R65> abs(const valarray<T, M>&);
template<class T, class M> valarray<T, R66> acos(const valarray<T, M>&);
template<class T, class M> valarray<T, R67> asin(const valarray<T, M>&);
template<class T, class M> valarray<T, R68> atan(const valarray<T, M>&);
template<class T, class M> valarray<T, R69> cos(const valarray<T, M>&);
template<class T, class M> valarray<T, R70> cosh(const valarray<T, M>&);
template<class T, class M> valarray<T, R71> exp(const valarray<T, M>&);
template<class T, class M> valarray<T, R72> log(const valarray<T, M>&);
template<class T, class M> valarray<T, R73> log10(const valarray<T, M>&);
template<class T, class M> valarray<T, R74> sin(const valarray<T, M>&);
```

```
template<class T, class M> valarray<T, R75> sinh(const valarray<T, M>&);
template<class T, class M> valarray<T, R76> sqrt(const valarray<T, M>&);
template<class T, class M> valarray<T, R77> tan(const valarray<T, M>&);
template<class T, class M> valarray<T, R78> tanh(const valarray<T, M>&);
```

1 If f be the name of one of these function and x the array argument to which they are applied, then their requirements and their effects are equivalent to those of apply(f, x).

valarray binary functions [lib.valarray.bfuncs]

1 An implementation must support void and any other storage model it introduces as substitutions for M, M1 and M2 in _lib.valarray.bfuncs_.

```
template<class T, class M1, class M2> valarray<T, R82>
  atan2(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<T, R83>
  atan2(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R84>
  atan2(const T&, const valarray<T, M>&);
template<class T, class M1, class M2> valarray<T, R85>
  min(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<T, R86>
  min(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R87>
  min(const T&, const valarray<T, M>&);
template<class T, class M1, class M2> valarray<T, R88>
  max(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<T, R89>
  max(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R90>
  max(const T&, const valarray<T, M>&);
template<class T, class M1, class M2> valarray<T, R91>
  pow(const valarray<T, M1>&, const valarray<T, M2>&);
template<class T, class M> valarray<T, R92>
  pow(const valarray<T, M>&, const T&);
template<class T, class M> valarray<T, R93>
  pow(const T&, const valarray<T, M>&);
```

1 If f be the name of one of these function and x and y the arguments to which they are applied, then their requirements and their effects are equivalent to those of apply(f, x, y).

Class slice [lib.class.slice]

```
namespace std {
  class slice {
  public:
    slice();
    slice(ptrdiff_t, ptrdiff_t, ptrdiff_t);

    ptrdiff_t start() const;
    ptrdiff_t length() const;
    ptrdiff_t stride() const;
  };
}
```

1 The slice class represents a BLAS-like slice from an array. Such a slice is specified by a starting index, a length, and a stride.

[Footnote:
[programs may instantiate this class.

slice constructors [lib.cons.slice]

```
slice();
slice(ptrdiff_t start, ptrdiff_t length, ptrdiff_t stride);
slice(const slice&);
```

1 The default constructor for slice creates a slice which specifies no elements. A default constructor is provided only to permit the declaration of arrays of slices. The constructor with arguments for a slice takes a start, length, and stride parameter.

2 [Example
slice(3, 8, 2) constructs a slice which selects elements 3, 5, 7, ... 17 from an array.
--- end example]

slice access functions [lib.slice.access]

```
ptrdiff_t start() const;
ptrdiff_t length() const;
ptrdiff_t stride() const;
```

1 These functions return the start, length, or stride specified by a slice object.