In ANSI public comment T29, Daveed Vandevoorde <vandevod@cs.rpi.edu> says:

> Comments on the proposed <valarray> header
> ...
> Probably the simplest way to address the above concerns is to simply
> abandon the standardization of a numerical array.

I would like to take this alternative seriously.

With the advent of Todd Veldhuizen’s work on Expression Templates, it is far from clear that valarray<> is the appropriate vehicle to aid in optimizing numeric array processing in C++. (For those who have not read Veldhuizen’s work in C++ Report, a copy may be found at <http://www.roguewave.com/>.) His work implies that using even a vendor-optimized/compiler-supported valarray<> may cost a factor of two or more in speed compared to using another library based on portable language facilities. This brings into question the value of the valarray<> template; the original argument in its favor was that it provided the hooks to permit optimal implementation "under the hood" (that’s "under the bonnet" for you Brits).

This is not a formal proposal to eliminate valarray<>, yet; it is instead a request for comments. I would like particularly to hear from ISO representatives whose vote might be forced to change if it is removed.

Nathan Myers
myersn@roguewave.com

Proposed Resolution:

Daveed Vandevoorde had a separate proposal (X3J16/96-0039, WG21/N0857) in the pre-Santa Cruz mailing. In Santa Cruz the library working group asked him to write up a proposal describing the exact changes that would need to be made to the draft to implement this proposal for Stockholm.

Requestor: Nathan Myers
Owner: Judy Ward
Emails: (email reflector messages that discuss this issue)
c++std-lib-3880
c++std-lib-3883
c++std-lib-3886
c++std-lib-3887
c++std-lib-3889
c++std-lib-3897
c++std-lib-3900
c++std-lib-3906
To: C++ libraries mailing list
Message c++std-lib-4427

I see we have a sqrt(complex) that returns a complex (of the right type). However, doesn’t a complex have MANY square roots?

Are anyone conducting a review of the math library?

- Bjarne

To: C++ libraries mailing list
Message c++std-lib-4430

Bjarne Stroustrup writes:

> I see we have a sqrt(complex) that returns a complex (of the right type). However, doesn’t a complex have MANY square roots?

Well, it has two. If x is a root, then -x is also a root. By widespread convention, the root with phase angle [-pi/2, pi/2) (as I recall) is preferred as the return value for sqrt.

> Are anyone conducting a review of the math library?

We’ve had some useful feedback from the heavy hitters in the C math library community.

P.J. Plauger

Proposed Resolution:

See Issue 26/016, specification of branch cuts and ranges.

Requestor: Bjarne Stroustrup
Owner: Judy Ward
Emails: (email reflector messages that discuss this issue)
Papers: (committee documents that discuss this issue)

26.2.1 I believe the term "norm" commonly refers to the square root of the squared magnitude (i.e. abs), and not the squared magnitude. Is a function for the squared magnitude needed? Note that the squared magnitude can be computed from abs with only deserved over/underflow, but not vice versa.

Proposed Resolution:

Discussed at Santa Cruz, the library working group suggested the name abs_sqr() but the full committee wanted to see what other languages used. I’ve looked in the FORTRAN and ADA standard and have not found an equivalent function.

Requestor: ?? (public comment)
Owner: Judy Ward

This resolution was from the TR mentioned above and also follows the conventions for branch cuts and ranges in the ADA95 standard and the FORTRAN 90 standard.

Expand Section 26.2.7 to:

Section 26.2.7 complex transcendental

template <class T> complex<T> acos (const complex<T>& x);
Notes: The branch cuts are outside the interval [-1,1] along the real axis.

Returns: the complex arc cosine of x, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [0,pi] along the real axis.

template <class T> complex<T> asin (const complex<T>& x);
Notes: The branch cuts are outside the interval [-1,1] along the real axis.

Returns: the complex arc sine of x, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [-pi/2, pi/2] along the real axis.

template <class T> complex<T> atan (const complex<T>& x);
Notes: The branch cuts are outside the interval [-i,i] along the imaginary axis where i is imag(x).
Returns: the complex arc tangent of x, in the range of a strip
mathematically unbounded along the imaginary axis and in the
interval [-pi/2, pi/2] along the real axis.

```cpp
template <class T> complex<T> atan2 (const complex<T>& x, const complex<T>& y);
template <class T> complex<T> atan2 (const complex<T>& x, T y);
template <class T> complex<T> atan2 (T x, const complex<T>& y);
```

Notes: The branch cuts are outside the interval [-1,1] along
the imaginary axis.

Returns: the complex arc tangent of y/x, in the range of a strip
mathematically unbounded along the imaginary axis and in the
interval [-pi, pi] along the real axis.

```cpp
template <class T> complex<T> cos (const complex<T>& x);
```

Returns: the complex cosine of x.

```cpp
template <class T> complex<T> cosh (const complex<T>& x);
```

Returns: the complex hyperbolic cosine of x.

```cpp
template <class T> complex<T> exp (const complex<T>& x);
```

Returns: the complex base e exponential of x.

```cpp
template <class T> complex<T> log (const complex<T>& x);
```

Notes: The branch cuts are along the negative real axis.

Returns: the complex natural (base e) logarithm of x, in the range of a strip
mathematically unbounded along the real axis and in the
interval [-i*pi, i*pi] along the imaginary axis where i is
imag(x).

```cpp
template <class T> complex<T> log10 (const complex<T>& x);
```

Notes: The branch cuts are along the negative real axis.

Returns: the common (base 10) logarithm of x.

```cpp
template <class T> complex<T> pow (const complex<T>& x, const complex<T>& y);
template <class T> complex<T> pow (const complex<T>& x, T y);
template <class T> complex<T> pow (T x, const complex<T>& y);
```

Notes: The branch cut for x is along the negative real axis.

Returns: the complex power of base x raised to the y-th power.

```cpp
template <class T> complex<T> sin (const complex<T>& x);
```

Returns: the complex sine of x.

```cpp
template <class T> complex<T> sinh (const complex<T>& x);
```

Returns: the complex hyperbolic sine of x.

```cpp
template <class T> complex<T> sqrt (const complex<T>& x);
```

Notes: The branch cuts are along the negative real axis.

Returns: the complex square root of x, in the range of
the right half-plane.
template <class T> complex<T> tan (const complex<T>& x);

Returns: the complex tangent of x.

template <class T> complex<T> tanh (const complex<T>& x);

Returns: the complex hyperbolic tangent of x.

Requestor: ?? (public comment)
Owner: Judy Ward
Emails: (email reflector messages that discuss this issue)
Papers: (committee documents that discuss this issue)

*****************************************************************************
Work Group: Library
Issue Number: 26/019
Title: Should typedefs be provided in complex lib?
Section: 26.2.7 Old
Status: active
Description:

Should the complex library provide typedefs for the
the specialization complex<float>, complex<double>,
and complex<long double> (like the string library
provides for basic_string<char> and basic_string<wchar_t>)?

Proposed Resolution:

Possible names for these typedefs are fcomplex, dcomplex, lcomplex
(or ldcomplex) or float_complex, double_complex, ldouble_complex
(or lddouble_complex).

Declarations (such as typedef complex<float> fcomplex) would have to be
added to the bottom of the Complex synopsis in Section 26.2 and after
each specialization in Section 26.2.2.

Requestor: Tom Plum and others
Owner: Judy Ward
Emails: (email reflector messages that discuss this issue)
Papers: (committee documents that discuss this issue)

*****************************************************************************
Work Group: Library
Issue Number: 26/020
Title: order of argument to valarray constructors
Section: 26.3 New
Status: active
Description:

- valarray constructors have (value, size_t num)
  normally in the library it is the other way round
  (string, vector, etc.).
  This should get fixed to be consistent.

Comment from Judy Ward:
valarray has:
valarray(const T& val, size_t n) initializes n elements of the array with val
valarray(const T* ap, size_t n) initializes first n elements with corresponding e
lements in array pointed to by ap

basic_string has:
basic_string(size_type n, charT c) // inconsistent with valarray
basic_string(const charT, size_type) // consistent with valarray
vector has:
vector(size_type n, const T& value) // inconsistent with valarray

So I think only the first constructor needs changing for consistency. This would mean in Section 26.3.1 and Section 26.3.1.1. change
valarray(const T&, size_t) to valarray(size_t, const T&)

Comment for Daveed Vandevoorde <vandevod@cs.rpi.edu>:
I agree that it is sufficient (and don’t care to much either way), but I would find it unintuitive to have the size parameter sometimes in the second and sometimes in the first position; at least within valarray I think it would be good to stay consistent (my personal preference also goes to size as the first argument to keep the vector<T> convention).

Comment from Nicolai on Daveed’s mail:
NO, i disagree.
First, consistence is a big goal.
Second, i would agree if the size parameter would have the same meaning. But is hasn’t. First it is "num times of ..." second it it "take ...", but only num elements of it". So it is OK to have different positions.

Proposed Resolution:
Change Section 26.3.1 and Section 26.3.1.1 change
valarray(const T&, size_t) to valarray(size_t, const T&)
In the description in Section 26.3.1.1 change "second" to "first" and "first" to "second".

Requestor: Nicolai Josuttis
Owner: Judy Ward
Emails: (email reflector messages that discuss this issue)
Papers: (committee documents that discuss this issue)

***************************************************************************

Work Group: Library
Issue Number: 26/021
Title: copy ctor declared for slice_array
Section: 26.3 New
Status: active
Description:
- In 26.3.3 Slices have no copy constructor but in 26.3.3.1 they have. What’s correct ?

Proposed Resolution:
On p. 17-7 it says "For the sake of exposition, Clause 18 through 27 do not describe copy constructors, assignment operators, or (non-virtual) destructors with the same apparent semantics as those that can be generated by default."

I think that is the situation here, so the declaration in 26.3.3.1 should be removed.

Requestor: Nicolai Josuttis
Owner: Judy Ward
Emails: (email reflector messages that discuss this issue)
Papers: (committee documents that discuss this issue)

***************************************************************************
Title: int should be size_t for element type of indirect_array
Section: 26.3 New
Status: active
Description:
- In 26.3.1.4 Index operator for indirect arrays has element type size_t, but in 26.3.8 it has int. What’s correct?

Comment for Daveed Vandevoorde <vandevod@cs.rpi.edu>:

This has proven to somewhat of a problem in actual valarray code: some algorithms expressed in terms of indirect access create negative intermediate value (e.g.: a[p+q] = 0.0; // p, q can contain negative values) requiring signed types (size_t can be unsigned). The current workaround requires a temporary valarray<size_t> or not using the indirect access mechanism.

My proposal would be to either use ints (probably too restrictive) or valarray<T>::index which would be required to be a signed integral type with at least the range of int.

Would it be acceptable to introduce a valarray<T>::index_type typedef? (or even make ‘int’ or ‘long int’ the index/stride type?)

Comment from Dave Dodgson:
Certainly we could do that, the question is to what do we set it? Perhaps we should make this a template parameter (with a default of int or long int). We would no longer need to include <cstddef> if we did this.

Proposed Resolution:

Section 26.3.1
Add:
// types:
typedef implementation_defined index_type;

Change:
T operator[](size_t) const;
T& operator[](size_t);
valarray<T> operator[](const valarray<size_t>&) const;
indirect_array<T> operator[](const valarray<size_t>&);
to:
T operator[](index_type) const;
T& operator[](index_type) const;
valarray<T> operator[](const valarray<index_type>&) const;
indirect_array<T> operator[](const valarray<index_type>&);

Section 26.3.1.3 and 26.3.1.4
change size_t to index_type

Section 26.3.8
change:
indirect_array<T> valarray<T>::operator[](const valarray<int>&)
to:
indirect_array<T> valarray<T>::operator[](const valarray<index_type>&)

Requestor: Nicolai Josuttis
Owner: Judy Ward
 Emails: (email reflector messages that discuss this issue)
c++-std-lib-4674
c++-std-lib-4675
c++-std-lib-4679
Papers: (committee documents that discuss this issue)
*****************************************************************************
Description:

- In 26.3.1 min()/max() are member functions, in 26.3.2.3 they are global. What's correct?

Comment for Daveed Vandevoorde <vandevod@cs.rpi.edu>:
Note that there are two min's (and two max's): one returning the smallest element in an array and one taking two arrays and returning an new array such that \((\text{min}(a, b))[i] = \text{min}(a[i], b[i])\). I suspect Kent Budge intended the member function to be the former and the regular function to be the latter. Personally, I rather keep this sort of function outside the class interface.

Resolution:

Add to Section 26.3.1.7:

T min() const;

Returns the smallest element in the array.

T max() const;

Returns the largest element in the array.

Section 26.3:
Change
-template <class T> T min(const valarray<T>&);  
template <class T> T max(const valarray<T>&);  
to:
-template <class T> valarray<T> min(const valarray<T>&, const valarray<T>&);  
template <class T> valarray<T> max(const valarray<T>&, const valarray<T>&);

Change Section 26.3.2.3
Change:
-template <class T> T min(const valarray<T>& a);  
template <class T> T max(const valarray<T>& a);  
to:
-template <class T> valarray<T> min(const valarray<T>& a, const valarray<T>& b);  
template <class T> valarray<T> max(const valarray<T>& a, const valarray<T>& b);

Copy the first paragraph into Section 26.3.1.7, since the same rule applies to the member functions min() and max().

change second paragraph to read:

The min() function returns an array such that \((\text{min}(a, b))[i] = \text{min}(a[i], b[i])\).
The max() function returns an array such that \((\text{max}(a, b))[i] = \text{max}(a[i], b[i])\).

Remove the third paragraph or move it to Section 26.3.1.7.

Requestor: Nicolai Josuttis
Owner: Judy Ward
Emails: (email reflector messages that discuss this issue)
Papers: (committee documents that discuss this issue)
- I see a problem if I do:
  \[
  \text{valarray<double> va;}
  \text{va *= 2;}
  \]
  As 2 is no double, no function is found because
  all templates only have one type and templates have restricted
  type conversions.
  Wouldn’t it make sense to use two template types, one for
  the elements in the valarray and one for the elements
  i operate with, to let this work?
  Example:
  \[
  \text{template <class T>}
  \text{class valarray {}
  \text{...
  \text{template <class T2>
  \text{valarray<T>& operator*= (const valarray<T2>&);}
  \text{template <class T2>
  \text{valarray<T>& operator*= (const T2&);}
  \text{...}
  \text{}}
  \text{}}
  \]

- One thing I missed really:
  \[
  \text{va[slice(3,4,2)] *= 2;}
  \]
  should be possible.
  Or in general, for all subset types assignment operators should be
  overloaded for one simple value on the right side (as it is
  for valarrays):
  \[
  \text{template <class T>}
  \text{class slice_array {}
  \text{...
  \text{template <class T2>
  \text{void operator*= (const valarray<T2>&); // see above}
  \text{template <class T2>
  \text{void operator*= (const T2&); // NEW !!!}
  \text{...}
  \text{}}
  \text{}}
  \]

Proposed Resolution:

Comment for Judy Ward:
I’m not sure if it’s a good idea to let users use arbitrary
 types for arithmetic operators .. for example would you
 want the compiler to let them add a char* to an valarray<int>?  
Also I think it might lead to ambiguities or wrong behaviour, i.e.:

\[
\text{valarray<double> vd;}
\text{slice_array<double> si;}
\text{vd += si;}
\]

Currently the only choice is to:
  use valarray(slice_array) ctor to create a valarray
  apply void operator*= (const valarray<T2>&) operator

With your proposal one could:
  instantiate a void operator*= (const slicearray<double>&)
  operator*=

I’m not positive if you would get an ambiguity error from the
 compiler or if it would choose the wrong thing (the second one).
Comment for Daveed Vandevoorde <vandevod@cs.rpi.edu>:
Indeed, the example that I showed (in pre-Tokyo discussions, I believe) is:

```c++
valarray<int> a;
valarray<valarray<int>> b;
// ...
b += a;
```

Is the latter a scalar assignment f a mixed-type vector-assignment?

Mixed-type operations really bring a lot of trouble (I tried to implement them --- I think valarray<Troy> 1.x may still have that feature --- but I found that it leads to extreme mess, e.g., when debugging numerical code).

Proposed Resolution:

Close this issue!

Requestor: Nicolai Josuttis
Owner: Judy Ward
Emails: (email reflector messages that discuss this issue)
Papers: (committee documents that discuss this issue)

*****************************************************************************
Work Group: Library
Issue Number: 26/025
Title: should STL-like semantics be added to valarray?
Section: 26.3 New
Status: active
Description:

- Perhaps it would be senseful to have as much container support as possible. At least begin(), end() and push_back() and insert() were VERY senseful for copying values into and out of a valarray (push_back() for back_inserter).

Comment for Daveed Vandevoorde <vandevod@cs.rpi.edu>:
begin() and end() are easy to specify (and I think they should indeed be added and defined as &a[0] and &a[0]+a.size() respectively), but valarray is specifically _not_ a dynamic array. So I don’t think operations that implicitly resize a valarray should be included.

Comment from Nicolai:
As Daveed wrote, push_back() and insert() are a problem as valarray is specifically _not_ a dynamic array.
The reason for the latter is to have an easy interface to create the arrays i want to do numerical stuff with.
At the moment I see only the chance to use a T* array or to set the values element by element.
But i think in practice reading some values and do some numerical operations would be a normal usage.
Or to write it in another form: What’s the best/normal way to prepare valarrays for numerical operations?

Proposed Resolution:

Section 26.3.1

Add:
Allocator argument to valarray
Allocator default arg to constructors
// types
(look in section 21.1.13 -- add all the typedefs)
from size_type to const_reverse_iterator)

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

Add new Section 26.3.1.
copy section 21.1.1.5 (substituting valarray instead of basic_string)

Requestor: Nicolai Josuttis
Owner: Judy Ward
Emails: (email reflector messages that discuss this issue)
Papers: (committee documents that discuss this issue)

*****************************************************************************

Work Group: Library
Issue Number: 26/026
Title: should sum() be a template?
Section: 26.3 New
Status: active
Description:

- sum() should be a template for function objects like accumulate.
  Or it may be even unnecessary. If begin() and end() would exist, you could use accumulate() then.

Comment for Daveed Vandevoorde <vandevod@cs.rpi.edu>:
The need for sum(...) is common enough to warrant its own function. However I agree (I submitted this during the CD1 public comment period) that a general "reduce(...)" function taking a functor would be nice. I also think "apply" should be modified in this way.

Proposed Resolution:
Add these as valarray non-member functions.

Add:

Section 26.3.2.5 valarray application functions

template<class T, class F> T reduce(const valarray<T> & a, const F& f);

F must be a function object for which the binary function-call operator()(x, y) is applicable when x and y are of type T. Let f(x, y) by 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).

template<class T> T sum(const valarray<T> & a);

The result of sum(a) is equal to reduce(a, std::plus). std::plus is described in Section 20.3.2 [lib.arithmetic.operations].

template<class T, class F> valarray<T> apply(const valarray<T>& a, const F& f);
F must be a function object for which a unary function-call operator\((x)\) exists and the function returns an array \(r\) such that \(r[i] == f(a[i])\).

```cpp
template<class T, class F>
valarray<T> apply(const valarray<T>& a, const valarray<T>& b, const F& f);
```

F must be a function object for which a binary function-call operator\((x,y)\) is applicable when \(x\) and \(y\) are of type \(T\). The function returns an array \(r\) such that \(r[i] == f(a[i], b[i])\).

Requestor: Nicolai Josuttis
Owner: Judy Ward
Emails: (email reflector messages that discuss this issue)
Papers: (committee documents that discuss this issue)

*****************************************************************************
Work Group: Library
Issue Number: 26/027
Title: should gslices be changed/removed?
Section: 26.3 New
Status: active
Description:

To: C++ libraries mailing list
Message c++std-lib-4673
Hello,

Since Santa Cruz I’ve been implementing most of the valarray functionality and now I have general slices as well (though they’re unacceptably inefficient... but that’s not the issue I would like to raise here).

Along with each valarray-feature, I also try to write a small program demonstrating a reasonable use for it. However, I could not find such a use for general slices. The current WP mentions that they are useful for the implementation of ‘’multidimensional arrays’’, but I found it far easier to implement those directly on top of valarrays.

Has anyone else used gslices in any practical way?

I attribute the problems I mention at least in part to the following:
- gslices are no valarrays and more limited in functionality
- gslices have no corresponding indexing scheme
- the ‘’number of dimensions’’ of a gslice is a run-time quantity, which seriously their use (must synthesize local valarrays)

Wrt. the last point I wonder if this stands in the way of direct compiler support?

Here are some of the solutions I can think of:
1) Do nothing: this is not harmful, but I expect no-one will seriously want to use gslices and they will thus be an unnecessary burden to implementors.
2) Drop gslices: this is not harmful either unless someone has already planned a serious project that requires them.
3) Replace the gslice functionality by multidimensional valarrays: Although I think multidimensional valarrays are what many really want, I don’t think anyone wants to work out a complete design in this round of standardization. However, I think a careful approach will allow a future extension in this sense.

I have a few more valarray issues that I hope to bring up in the next few weeks, but this one seemed like a good start ;-)
Daveed

Proposed Resolution:

Drop gslice and gslice_array -- Remove Sections 26.3.5 and 26.3.6

Remove functions that use gslice_array in valarray section 26.3.1 (i.e. do a search for "gslice" and remove everything)

Requestor: Daveed Vandevoorde
Owner:        Judy Ward
Emails: (email reflector messages that discuss this issue)
Papers: (committee documents that discuss this issue)

*****************************************************************************
Work Group:     Library
Issue Number:   26/028
Title: rename valarray::length() to valarray::size()
Section:        26.3 New
Status:         active
Description:

I propose to rename:

    size_t length() const;

to:

    size_t size() const;

to keep consistency with other container-like things.

Proposed Resolution:

Search for all instances of "length" in Section 26.3 and change it to "size".

Requestor: Daveed Vandevoorde
Owner:        Judy Ward
Emails: (email reflector messages that discuss this issue)
Papers: (committee documents that discuss this issue)

*****************************************************************************
Work Group:     Library
Issue Number:   26/029
Title: valarray::operator|| and valarray::operator&&
Section:        26.3 New
Status:         active
Description:

Should operator|| and operator&& really be overloaded for arrays? If yes, shouldn’t the return-type be an array of bool?

My proposal: drop operator|| and operator&& since the short-circuit principle cannot be emulated for user-defined types.

Proposed Resolution:

Remove operator|| and operator&& from 26.3.2.1
OR change the return type to valarray<bool>

Requestor: Daveed Vandevoorde
Owner:        Judy Ward
Work Group: Library
Issue Number: 26/030
Title: fix up what headers are included by complex, valarray, and numeric
Section: 26.3 and 26.4 New
Status: active
Description:
These headers do not specify what other C++ headers they must include.

Proposed Resolution:

In the synopsis for complex (26.2), add:
#include <iosfwd>
In the synopsis for valarray (26.3), add:
#include <cstdlib>
In the synopsis for numeric (26.4) add:
#include <utility>
#include <iterator>

Requestor: Judy Ward
Owner: Judy Ward

Work Group: Library
Issue Number: 26/031
Title: should valarray unary ops be non-members?
Section: 26.3 New
Status: active
Description:
I noticed that valarrays treat unary operators as member functions, whereas complex treats them as regular functions.

Proposed Resolution:
I think it is better to have them be regular functions since normal conversions could be applied.

Remove the unary operator declarations from inside the class valarray in Section 26.3.1.

Move Section 26.1.3.5 to Section 26.3.2 (possibly 26.3.2.1?) replacing the decls:

valarray<T> operator+() const;
valarray<T> operator-() const;
valarray<T> operator~() const;
valarray<T> operator!() const;

with:

template <class T> valarray<T> operator+(const valarray<T>& lhs) const;
template <class T> valarray<T> operator-(const valarray<T>& lhs) const;
template <class T> valarray<T> operator~(const valarray<T>& lhs) const;
template <class T> valarray<T> operator!(const valarray<T>& lhs) const;
(this one might have to be changed to return valarray<bool> see Issue 26/032)

Requestor: Daveed Vandevoorde
Work Group: Library
Issue Number: 26/032
Title: Should valarray::operator! return valarray<bool> not valarray<T>?
Section: 26.3 New
Status: active
Description:

Proposed resolution:

Change return type of valarray::operator! in Section 26.3 to valarray<bool>.

Requestor: Daveed Vandevoorde
Owner: Judy Ward
Emails: (email reflector messages that discuss this issue)
Papers: (committee documents that discuss this issue)

******************************************************************************

Work Group: Library
Issue Number: 26/033
Title: clarify definition of boolean mask subset operator
Section: 26.3 New
Status: active
Description:

Another valarray issue. The ‘boolean mask subset operator’ (operator[] taking an array of bools) currently has a somewhat bizarre definition (depending on whether it is applied to a const or to a non-const array). I suspect that what was really aimed for was ‘assignment masking’ (because some architectures indeed have hardware to mask operations on a per element basis), i.e. the current semantics when the operator is applied to a non-const array.

Proposed resolution:

To resolve and clarify this issue, I propose two measures:

1) drop the const member-operator[](const valarray<bool>&);

2) rename the non-const version to ‘mask(const valarray<bool>&)’ to emphasize the different character of this function compared to the subset-selectors.

In Section 26.3.1 and Section 26.3.1.4 remove:
valarray<T> operator[](const valarray<bool>&) const;
mask_array<T> operator[](const valarray<bool>&) const;

Add to Section 26.3.1:
mask_array<T> mask(const valarray<bool>&);

Add to Section 26.3.1.7:
mask_array<T> mask(const valarray<bool>& v);

This function returns an object of type mask_array<T> with reference semantics to the *this array. The elements of *this at positions i
for which v[i] == false will be masked off when performing assignments  
and computed assignments to the returned object.

2 The behavior is undefined if this->size() != v.size().

Requestor: Daveed Vandevooorde
Owner:         Judy Ward
Emails: (email reflector messages that discuss this issue)
Papers: (committee documents that discuss this issue)

***********************************************************************