Fixing `valarray` for Real World Use

AFNOR

The `valarray` library is designed to provide efficient BLAS-like capabilities for value array processing. However, we have identified five major flaws in its specification, preventing effective and efficient use in real world programs. The following considerations are based on feedback from "numericists" using that library in their projects. For each defect we suggest what appear to us to be the most acceptable resolutions by the practioners.

1 There ought to be a way of iterating over a **`valarray`**

This issue is also known as *issue* 77. `valarray` is designed primarily for numerical computing. As such it has to cooperate with legacy code around — mostly written in C or FORTRAN. The unfortunate signature

```
T valarray<T>::operator[](size_t) const
```

does rule out the possibility to pass a `const valarray` content data to an existing C function for inspection. That severe rectriction is unnecessary and does, in fact, limit `valarray` usability. We strongly feel that that disservices `valarray`'s primary purpose. The complains and argumentations we get from "numericists" who use `valarray` in their jobs tend to support the view that this unfortunate interface specification is defectuous. We suggest to change that signature to

```
const T& valarray<T>::operator[](size_t) const
```

Not only will that change make `valarray` usable in the real world, but it will also contribute to integrate the valarray library to the rest of the standard library without (necessarily) turning `valarray` into a standard container. The "free aliasness" property (26.3.1/2) is a contract, a requirement that the programmer follows; it doesn't need to be expressed in such an unnecessary restrictive form (26.3.2.3). The sole mention of 26.3.1/2 does suffice to grant an implementor the right to use hardware optimization.

2 A `valarray` ought to be constructible from helper classes `slice_array`, `gslice_array`, `mask_array`, `indirect_array`

Consider

```
#include <valarray>
int main()
{
  using namespace std;
  valarray<double> v(10, 1.0);
  valarray<double> w = v[slice(0, 5, 2)];   //#1
}
```

The line marked `#1` is invalid since the copy constructor

```
slice_array<T>::slice_array<T>(const slice_array<T>&)
```

is inaccessible. The same is true for the other valarray helper classes. We strongly feel that that is a defect. We suggest to make public those copy constructors. Furthermore, we suggest to make public the corresponding copy and assignment operators as well, for reasons exposed below.

3 Valarray helper classes need to be user instantiable

Indeed, BLAS programs manipulate `valarray`s mostly through array slices (indexing with `slice`, `gslice`, `valarray<bool>` or `valarray<size_t>`); therefore they need to name the resulting objects. Here is an typical question we got from users:

> I need to use `slice` and `gslice` (for tensors of course) and indirection (for lots of linear algebra stuff)–I don't imagine I'll very often want to refer to a `valarray` other than through slices and indirection. I'm sure you've faced this situation too. What do you do?

Allowing users to instantiate helper classes remedies to this defect, without preventing or complicating possible optimization.

4 Helper classes `slice_array`, `gslice_array`, `mask_array`, `indirect_array` ought to have reference semantics

The helper classes `slice_array`, `gslice_array`, `mask_array`, `indirect_array` were designed to have reference semantics. But in *practice* that is only partially true. For they lack several important member functions necessary for (indexed) array processing, such as:

```
T& _array<T>::operator[](size_t);
const T& _array<T>::operator[](size_t) const;

_array<T>& _array<T>::operator=(const slice_array<T>&);
_array<T>& _array<T>::operator=(const gslice_array<T>&);
_array<T>& _array<T>::operator=(const mask_array<T>&);
_array<T>& _array<T>::operator=(const indirect_array<T>&);

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

_array<T>& _array<T>::operator*=(const T&);
_array<T>& _array<T>::operator/=(const T&);
_array<T>& _array<T>::operator%=(const T&);
_array<T>& _array<T>::operator+=(const T&);
_array<T>& _array<T>::operator-=(const T&);
_array<T>& _array<T>::operator^=(const T&);
_array<T>& _array<T>::operator&=(const T&);
_array<T>& _array<T>::operator|=(const T&);
_array<T>& _array<T>::operator<<=(const T&);
_array<T>& _array<T>::operator>>=(const T&);

T sum() const;
T min() const;
T max() const;
```

```
valarray<T> shift(int) const;
valarray<T> cshift(int) const;
valarray<T> apply(T func(T)) const;
valarray<T> apply(T func(const T&)) const;
```

We stress on the practical fact these member functions are needed in BLAS. We suggest their adoption. Their absence is a major handicap for use in effective and efficient BLAS programs.

5 Non-member functions working with `valarrays` ought to work with helper classes `slice_array`, `gslice_array`, `mask_array`, `indirect_array`

This issue is a logical consequence of the previous requests. It was also raised by Nico Josuttis as *issue* 93. We do feel that this is indeed a defect. We therefore suggest that any standard non-member function taking a `const valarray&` as a parameter be overloaded to accept a `const helper class&`.