Doc. no. WG21/N1388
J16/02-0046
Date: 07 September 2002
Reply-To: Gabriel Dos Reis
INRIA Sophia Antipolis
2004 route des Lucioles — BP 79
06902 Sophia-Antipolis — France
Fax: 334 92 38 79 78
Email: gdr@acm.org

Enhancing Numerical Support

1 Introduction

C++ support for numerical computations, mostly for the basic algebraic datatypes, is suboptimal and somehow inefficient when compared to competing languages like Fortran or C99. Even more, wordings in the Standard are insufficient to facilitate inter-operation with those languages. The present paper makes proposals to address some of the pressing issues.

2 Complex numbers

2.1 Issues

2.1.1 `std::complex<>` over-encapsulated

The standard library `<complex>` component suffers from some form of over-abstraction described in the document [2]. In a nutshell, the issues are

- absence of explicit layout decription,
- no way to access individual parts as lvalues.

The absence of explicit description of `std::complex<T>` layout makes it imposible to reuse existing software developed in traditional languages like Fortran or C with unambigous and commonly accepted layout assumptions. There ought to be a way for practitioners to predict with confidence the layout of `std::complex<T>` whenever `T` is a numerical datatype.

The absence of ways to access individual parts of a `std::complex<T>` object as lvalues unduly promotes severe pessimizations. For example, the only way to change, *independently*, the real and imaginary parts is to write something like

```
complex<T> z;
// ...
// set the real part to 'r'
z = complex<T>(r, z.imag());
// ...
// set the imaginary part to 'i'
z = complex<T>(z.real(), i);
```

At this point, it seems appropriate to recall that a complex number is, in effect, just a pair of numbers with no particular invariant to maintain. Existing practice in numerical computations has it that a complex number datatype is usually represented by Cartesian coordinates. Therefore the over-encapsulation put in the specification of `std::complex<>` is not justified.

2.1.2 Why can't there be a complex key?

Practice with `std::complex<>` and the associative containers occasionally reveals artificial and distracting issues with constructs resembling:

```
std::set<std::complex<double> > s;
```

The main reason for the above to fail is the absence of an approriate definition for `std::less<std::complex<T> >`. That in turn comes from the definition of the primary template `std::less<>` in terms of `operator<`. The usual argument goes as follows: Since there is no ordering over the complex field *compatible* with field operations it makes little sense to define a function `operator<` operating on the datatype `std::complex<T>`. That is fine. However, that reasoning does not carry over to `std::less<T>` which is used, among other things, by associative containers as an ordering useful to meet complexity requirements.

2.2 Proposed resolutions

2.2.1 `std::complex<>` made concrete

The document [2] proposed to adopt C99 definition for `std::complex<>` in addition of the notion of *Enhanced POD*. In this paper, I propose an alternate resolution that solves the layout issue without the Core Language extension of Enhanced POD. Add the following requirements to 26.2 as 26.2/4:

> If `z` is an lvalue expression of type *cv* `std::complex<T>` then
>
> — then expression `reinterpret_cast<`*cv* `T(&)[2]>(z)` is well-formed; and

— `reinterpret_cast<`*cv* `T(&)[2]>(z)[0]` designates the real part of `z`; and

— `reinterpret_cast<`*cv* `T(&)[2]>(z)[1]` designates the imaginary part of `z`.

Moreover, if `a` is an expression of pointer type *cv* `complex<T>*` and the expression `a[i]` is well-defined for an integer expression `i` then

— `reinterpret_cast<`*cv* `T*>(a)[2*i]` designates the real part of `a[i]`; and

— `reinterpret_cast<`*cv* `T*>(a)[2*i+1]` designates the imaginary part of `a[i]`.

The first set of requirements makes it possible to efficiently access individual parts of a `std::complex<T>`; such accesses are then easily encoded in the library by having `real()` and `imag()` return references and not values. Example:

```
template<typename T>
  inline T& real(complex<T>& z)
  {
    return reinterpret_cast<T(&)[2]>(z)[0];
  }
template<typename T>
  inline const T& real(const complex<T>& z)
  {
    return reinterpret_cast<const T(&)[2]>(z)[0];
  }
template<typename T>
  inline T& imag(complex<T>& z)
  {
    return reinterpret_cast<T(&)[2]>(z)[1];
  }
template<typename T>
  inline const T& imag(const complex<T>& z)
  {
    return reinterpret_cast<const T(&)[2]>(z)[1];
  }
```

This solution has been tested with all current major implementations of the standard library and shown to be working — one particular implementation seems to be checking array bounds in a `reinterpret_cast`; that implementation however accepts pointer arithmetic and behaves as expected.

2.2.2 `std::complex<>` keying associative containers

To resolve the ordering issue mentioned in section 2.1.2 I propose to add the following partial specializations:

```
template<typename T>
  struct less<complex<T> >
     : binary_function<complex<T>, complex<T>, bool> {
     bool operator()(const complex<T>& z,
                     const complex<T>& w) const
     {
        return less<T>()(real(z), real(w))
                 || (!less<T>()(real(w), real(z))
                     && less<T>()(imag(z), imag(w)))
     }
  };
template<typename T>
  struct less_equal<complex<T> >
     : binary_function<complex<T>, complex<T>, bool> {
     bool operator()(const complex<T>& z,
                     const complex<T>& w) const
     {
        return !less<T>()(w, z);
     }
  };
template<typename T>
  struct greater<complex<T> >
     : binary_function<complex<T>, complex<T>, bool> {
     bool operator()(const complex<T>& z,
                     const complex<T>& w) const
     {
        return less<T>()(w, z);
     }
  };
template<typename T>
  struct greater_equal<complex<T> >
     : binary_function<complex<T>, complex<T>, bool> {
     bool operator()(const complex<T>& z,
                     const complex<T>& w) const
     {
        return !less<T>()(z, w);
     }
  };
```

3 Numerical array

3.1 A `const` and a value issue

Consider the following program:

```
#include <iostream>
#include <ostream>
#include <vector>
#include <valarray>
#include <algorithm>
#include <iterator>
template<typename Array>
  void print(const Array& a)
  {
    using namespace std;
    typedef typename Array::value_type T;
    copy(&a[0], &a[0] + a.size(),
         ostream_iterator<T>(std::cout, " "));
  }
template<typename T, unsigned N>
  unsigned size(T(&)[N]) { return N; }
int main()
{
  double array[] = { 0.89, 9.3, 7, 6.23 };
  std::vector<double> v(array, array + size(array));
  std::valarray<double> w(array, size(array));
  print(v);              // #1
  std::cout << std::endl;
  print(w);              // #2
  std::cout << std::endl;
}
```

While the call numbered `#1` succeeds, the call numbered `#2` fails because the const version of the member function `valarray<T>::operator[](size_t)` returns a value instead of a const-reference. That seems to be so for no apparent reason, no benefit. Not only does that defeats users' expectation but it also does hinder existing software (written either in C or Fortran) integration within programs written in C++. This issue is also described in section 1 of the document [1].

3.2 Proposed resolution

There is no reason why subscripting an expression of type `valarray<T>` that is const-qualified should not return a `const T&`. Not returning a non-

reference creates more problems than it solves any supposed concern. I propose to change the member function

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

to

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

This proposal increases usability within C++, software reuse and interoperability with existing libraries written in languages like Fortran and C99.

This proposal is already implemented and used in the GNU implementation of the C++ standard library.

Acknowledgments

I would like to thank Benjamin Kosnick (bkoz@redhat.com) for helpful comments and encouragments in preparing this paper. I'm also grateful to contributors of the French speaking usenet group `fr.comp.lang.c++` for testing and commenting on the sample code shown in the proposal describing `std::complex<>` layout.

References

[1] AFNOR, *Fixing `valarray` for Real World Use*, document WG21/N1246.

[2] R.W. Grosse-Kunstleve & D. Abrahams, *Predictable data layout for certain non-POD types*, document WG21/N1356.