Doc. no. N1612=04-0052
Date: March 29th, 2004
Reply-To: Thorsten Ottosen, nesotto@cs.auc.dk or tottosen@dezide.com

How we might remove the remaining shortcomings of `std::complex<T>`

The language of pairs is incorrect for Complex Arithmetic; it needs the Imaginary type.
Professor William Kahan

Introduction

Recently there have been written two proposals that seek to enhance the use of `std::complex<T>` [GKA02] [Rei02]. The proposals have my full support, but a few other changes could be made for the sake of elegant code, efficiency and intuitive behavior. The changes rely mainly on the introduction of an `imaginary` class describing imaginary numbers like $5i$. It is worth noticing that C99 has added support for complex numbers and described recommendations that seek to address these issues [C9903, 183ff]. One way to characterize this proposal is that it suggests partial support for Language Independent Arithmetic Part 3 (LIA-3) [ISO01].

Contents

1 The problem

As discussed in [Mar03], the lack of imaginary literals in C++ gives rise to both inelegant usage and efficiency concerns. To avoid this problem, the programmer may have to decompose complex arithmetic expressions into separate computations of real and imaginary parts thereby forgoing some of the advantages of compact notation. To illustrate the inelegant syntax consider how we code $(4 + 2i)/i$:

```
typedef complex<double> complex_t;
complex_t c3 = complex_t( 4, 2 ) / complex_t( 0, 1 );
```

The inefficiency occurs especially (but not exclusively) with multiplication and division:

```
complex_t z( 1, 2 );
z *= complex_t( 0, 1 ); // note: rhs is imaginary
z /= complex_t( 0, 1 ); // note: ditto
```

Complex multiplication normally takes 4 multiplications and 2 additions, but when a pure imaginary number is involved, multiplication boils down to 1 negation and 2 multiplications due to the zero real part [C9903, 184]. (Remark: an implementation might simply check for the zero real part, but neither Dinkumware nor STLport 4.6.1 does it. On Intel + win32 I tested the overhead of such a branching and it was hard to measure. I suspect that other platforms behave differently.) For division we normally need one comparison, 3 divisions, 3 multiplications and 3 additions instead of 2 divisions and 1 negation.

As a third concern, there is an unintuitive distinction between how real and imaginary numbers behaves when combined with complex numbers. For some applications it is important that complex-real and complex-imaginary mixed arithmetic does not promote real and imaginary numbers to complex numbers before a multiplication. However, that is impossible without imaginary numbers. This avoids the following problem with infinities [C9903, 47f]:

```
const double inf = numeric_limits<double>::infinity();
complex_t x      = 2.0 * complex_t( 3.0, inf );
complex_t y      = complex_t( 2.0, 0 ) * complex_t( 3.0, inf );
cout << x << y; // should print '(+6,+inf) (+nan,+inf)'
```

The first output is better since it maintains the right quadrant of the number. A similar problem exists with signed zeros:

```
x = 2.0 * complex_t( 3.0, -0 );
y = complex_t( 2.0, 0 ) * complex_t( 3.0, -0 );
cout << x << y; // should preferably print '(+6,-0) (+6,+0)'
```

The first result is desirable (because it preserves the right sign information), but none of my three C++ compilers will compute it because the standard does not specify the necessity of signed zeros [KD98, 15f]. This again leads to non-intuitive examples such as

```
complex_t z = -1; // or any other negative real
cout << sqrt( conj( z ) ) << conj( sqrt( z ) ) );
// prints: (+0,+1) (+0,-1)
```

The problem is that the equality $sqrt(conj(z)) = conj(sqrt(z))$ holds for certain complex numbers, but not for others. Further discussion of the importance of signed zeros can be found in [Gol91, 201f] and [Kah87].

2 The solution

The issues that arise because we do not have unsigned zeros cannot be resolved through a library proposal. The language needs to change, and it is an open question how this might be done. One possibility could be to require signed zeros if the hardware supports it.

However, all the remaining issues can be dealt with by a library extension: we simply need to add a new class that defines imaginary numbers and which interacts with the real and complex numbers. This small sample shows how we might use the new complex and imaginary classes:

```
typedef imaginary<double> imaginary_t;
complex_t z = 4.2 + 3.0*i;
complex_t x = 42.*i;
complex_t y = ( 4 + 2.0*i ) / ( 1.*i );
z *= 32.*i;
z /= 1.*i;
imaginary_t im( 4 );    // ok
imaginary_t im2 = 4;    // error, rhs is a real
imaginary_t im3 = 4.*i  // ok, rhs is now imaginary<double>
imaginary_t im4 = 5.*i + im * 4. - im3;
sqrt( im );              // error
sqrt( complex_t( im ) ); // ok
```

So the idea is simply to overload operators such that any interaction with `i` creates an imaginary number.

3 Changes to `<complex>`

This proposal suggests that the `<complex>` header should be extended to include the following:

```
template< typename T >
class imaginary
{
    imag_; // exposition only
public:
    explicit imaginary( T imag = 0 ); // explicit forces ’imaginary z = 2.*i’
    T value() const;
    imaginary& operator=( T r );
    imaginary& operator=( imaginary r );
    imaginary& operator+=( imaginary r );
    imaginary& operator-=( imaginary r );
    imaginary& operator*=( T r );
    imaginary  operator-() const;
    // note: no operator/() since result is complex
    // note: default copy operations are ok
};


struct imaginary_unit_t // one constant that works will all fp types
{
  imaginary_unit_t() {}
};

const imaginary_unit_t i;

template< typename T >
class complex
{
    // as before ... but
    // new constructors
    complex( imaginary<T> r );
    complex( T realval, imaginary<T> imagval );

    // new arithmetic
    complex& operator=( imaginary<T> r );
    complex& operator+=( imaginary<T> r );
    complex& operator-=( imaginary<T> r );
    complex& operator*=( imaginary<T> r );
    complex& operator/=( imaginary<T> r );
};

/////////////////////////////////////////////////////
// imaginary literals, ex: i * 4.2f, 4.3 * i

template< typename T >
imaginary<T> operator*( T l, imaginary_unit );
template< typename T >
imaginary<T> operator*( imaginary_unit, T r );
//
// ... also for operator +, -, /
//

template< typename T >
complex<T> operator*( const complex<T>& l, imaginary_unit );
```

```
template< typename T >
complex<T> operator*(  imaginary_unit, const complex<T>& r );
//
// ... also for operator +,-,/
//

//////////////////////////////////////////////////////
// imaginary arithmetic

template< typename T >
imaginary<T> operator+( imaginary<T> l, imaginary<T> r );
template< typename T >
imaginary<T> operator-( imaginary<T> l, imaginary<T> r );
template< typename T >
T operator*( imaginary<T> l, imaginary<T> r ); // note conversion to T
template< typename T >
imaginary<T> operator/( imaginary<T> l, imaginary<T> r );

//////////////////////////////////////////////////////
// mix-mode arithmetic: real-imaginary

template< typename T >
complex<T> operator+( T l, imaginary<T> r );
template< typename T >
complex<T> operator+( imaginary<T> l, T r );
template< typename T >
complex<T> operator-( T l, imaginary<T> r );
template< typename T >
complex<T> operator-( imaginary<T> l, T r );
template< typename T >
imaginary<T> operator*( T l, imaginary<T> r ); // note conversion to imaginary<T>
template< typename T >
imaginary<T> operator*( imaginary<T> l, T r ); // note conversion to imaginary<T>
template< typename T >
complex<T> operator/( T l, imaginary<T> r );
template< typename T >
complex<T> operator/( imaginary<T> l, T r );

//////////////////////////////////////////////////////////////
// mixed-mode arithmetic: complex-imaginary

template< typename T >
complex<T> operator+( const complex<T>& l, imaginary<T> r );
template< typename T >
complex<T> operator+( imaginary<T> l, const complex<T>& r );
template< typename T >
complex<T> operator-( const complex<T>& l, imaginary<T> r );
template< typename T >
complex<T> operator-( imaginary<T> l, const complex<T>& r );
template< typename T >
complex<T> operator*( const complex<T> l, imaginary<T> r );
template< typename T >
complex<T> operator*( imaginary<T> l, complex<T> r );
template< typename T >
```

```
complex<T> operator/( const complex<T>& l, imaginary<T> r );
template< typename T >
complex<T> operator/( imaginary<T> l, const complex<T>& r );

//////////////////////////////////////////////////////////////
// comparison

template< typename T >
bool operator==( imaginary<T> l, imaginary<T> r );
template< typename T >
bool operator!=( imaginary<T> l, imaginary<T> r );
template< typename T >
bool operator<( imaginary<T> l, imaginary<T> r );
//
// ... and also for >,>=,<=
//
template< typename T >
bool operator==( const complex<T> l, imaginary<T> r );
template< typename T >
bool operator==( imaginary<T> l, complex<T> r );
//
// ... and also for !=
//
```

4 Discussion

There are several issues of the implementation that should be discussed.

1. Should complex math functions be overloaded for imaginary types? It is probably not worth the trouble since implementation can already treat a real part of zero special if it is really beneficial.

2. Why should we provide a `imaginary_unit_t`? Would it not be easier to have `const imaginary<double> i;`? The problem is that we now have hardcoded `double` into the constant so we would need three constants. If the type of `i` is public available, one can also call it `j` which is common engineering practice. (Remark: C99 has the `I` macro.)

3. Should implicit conversion between `imaginary<T>` and `imaginary<U>` be allowed by providing templated conversions operators, extra template argument on `operator+()` etc.? The downside would be that narrowing conversions could suddenly happen implicit; however, this could probably be prohibited using the enable-if technique [JJL03]. Strinct conformance to the same value type, on the other hand, is good for performance.

4. One reviwer suggested the name `imaginary_symbol` instead of `imaginary_unit_t`.

5. How advanced should the constant `i` be? The implementation described here requires that `i` is always used in connection with the multiplication operator. It is (or at least will be) possible to make `i` itself more context aware, but it will require a much more elaborate implementation. The simple multiplication mechanism means that imaginary number are created the same way in C99 and C++.

5 Acknowledgements

Thanks to Andy Little, Daniel Frey and Guillaume Melquiond for their comments. Thanks to Walter Bright for making his critique of C++ complex numbers public.

References

[C9903] Rationale for International Standard—Programming Languages—C. http://anubis.dkuug.dk/jtc1/sc22/wg14/www/C99-RationaleV5.10.pdf, 2003. 1, 2

[GKA02] R.W. Grosse-Kunstleve and D. Abrahams. Predictable data layout for certain non-pod types, document wg21/n1356., 2002. 1

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. http://citeseer.nj.nec.com/goldberg91what.html, 1991. 3

[ISO01] Information technology—language independent arithmetic, part 3. http://std.dkuug.dk/jtc1/sc22/wg11/docs/n476.pdf, 2001. 1

[JJL03] Jeremiah Willcock Jaakko Järvi and Andrew Lumsdaine. enable_if. http://www.boost.org/libs/utility/enable_if.html, 2003. 6

[Kah87] William Kahan. Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit. The State of the Art in Numerical Analysis (eds. Iserles and Powell), 1987. 3

[KD98] Prof. W. Kahan and Joseph D. Darcy. How Java's Floating-Point Hurts Everyone Everywhere, 1998. 3

[Mar03] Digital Mars. D Complex Types and C++ std::complex . http://www.digitalmars.com/d/cppcomplex.html, 2003. 2

[Rei02] Gabriel Dos Reis. Enhancing numerical support, document wg21/n1388, 2002. 1