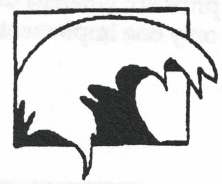
WG14/N313
X3J11/93-060
X3J16/93-0165
WG21/N0372

A Proposal for Standard C++ Complex Number Classes

Allan Vermeulen
Rogue Wave Software, Inc.
Technical Report 93-011



P.O. Box 2328
Corvallis, OR 97339
(503) 754-3010

1 Revision history

Revision 93-011a

- First Revision

2 Introduction and rationale

The lack of built in support for complex numbers is a major reason that few numerical codes are written in C. Fortunately, in C++ this impediment can be completely overcome by adding complex number types, easily implemented as classes, to the language's standard library. This document is a proposal for these standard types.

This proposal is based largely on two sources: the AT&T implementation of double precision complex numbers, and the proposed C extension described by David Knaak in "Complex extensions to C", document X3J11/92-075.

The classes proposed here are as compatable as possible with the types proposed in the C language extension. This implies that implementation of the C++ classes can be done using the native C types, and also that only one set of syntax and semantics need be learned for people programming in both C and C++.

A sample definition for the classes described here is given at the end of this document. You may find it helpful to refer to this while reading through this proposal.

2.1 Exclusion of imaginary types

It has been proposed (Jim Thomas, "Complex C extensions", document X3J11.1/93-012) that, in addition to complex types, types be added for imaginary numbers. This would allow improved semantics for operators between imaginary numbers and complex numbers when signed zeros, NANs (IEEE Not A Number), or infinities occur. It also has a nice, symmetric, feel — after all, the real part of a complex number has a representation, why not the imaginary part?

We do not include imaginary types in this proposal for two reasons. One, it increases the complexity of the design: three new types are added, and operators must be defined for all permutations of these new types with real types and complex types (for a description of the complexities see David Knaak, "What to do about the complex extension to C?", document X3J11/93-016). The second objection is a relative lack of prior art: whereas there is a great deal of experience with a complex class there is only one implementation (that the author is aware of) of an imaginary class.

3 Complex types

We propose adding three standard types: float_complex, double_complex, and long_double_complex. Each represents a complex number whose real and imaginary parts have the same characteristics as the corresponding scalar types. In practice, this requires that the implementation of the complex types be Cartesian, as opposed to, for example, polar. This selection of types, and the requirement that the implementation be Cartesian, corresponds with the proposed C language extension.

3.1 Types or classes?

The most straightforward implementation of the complex number types is as C++ classes. It is not required, however, that the types be implemented this way. They could be implemented, either for efficiency reasons or symmetry with C, as built in types.

- This implies that using the complex types as base classes is not allowed by the standard.

3.2 Including these types

The complex number classes, and all their associated functions, are defined by including the header file `complex.h`. Once the complex types are declared, the preprocessor macro `__STD_COMPLEX` will be defined.

- This provides a method for a program to differentiate between these new complex types and a `complex.h` which declares another set of complex classes, for example the AT&T complex implementation.
- In a simple class implementation, this macro could be used as a "wrapper" to prevent multiple inclusions of the header file.

4 Type conversion, initialization, and assignment

4.1 Type conversion

The following type conversions may take place:

- scalar → complex. Type conversion from a float, double, or long double to the corresponding complex type may occur. The resulting complex number has zero imaginary part.
- lower precision complex → higher precision complex. Automatic conversion from a lower to a higher precision complex type may occur.

Type conversion from higher to lower precision complex numbers is not done automatically. Instead, global functions are provided to do this conversion. The function name is the same as the target class name prepended with an underscore. The function takes a single const reference argument of the source type. Thus, to convert from double_complex to float_complex, use the function:

```
float_complex _float_complex(const double_complex&)
```

- It would have been preferable to eliminate these special functions and do narrowing conversion in the same way as for the built in types. Unfortunately, this approach leads to a couple of problems. First, any self-respecting C++ compiler will warn you if an implicit conversion from a built in type to a narrower type (e.g. double→float) takes place. It is unreasonable to expect such warnings from a class based implementation of complex numbers. Second, if both conversions exist, code such as the following:

```
void foo( float_complex a, double_complex b)
{
        a+b;
}
```

causes ambiguities without explicit casting.

- The conversion operators pass the parameter by const reference for symmetry with the standard C++ copy constructor. By contrast, the intrinsic functions defined later pass their parameter by value.

4.2 Initialization

Any allowed type conversion to a complex is also allowed as an initializer.

Copies of complex numbers can be created. This corresponds to a copy constructor in a class based implementation.

A complex number can be initialized from an ordered pair of two scalars representing the real and imaginary part of the complex number. The syntax is the standard one that would be expected:

```
double_complex x(3.4,2.3); // x is 3.4 + 2.3i
double_complex y = x - double_complex(2,3);
```

4.3 Assignment

An assignment operator is provided. As with the built in types, the type of an assignment expression is that of its left operand.

5 Arithmetic operators

All parameters for overloaded operators are passed by value, as in the existing AT&T class based implementation. The precise formulas used to implement complex arithmetic are implementation dependent. Reponse to errors is also implementation dependent.

The assignment operators, +=, -=, *=, /=, are defined where the left hand side is a complex type, and the right hand side is either a complex or a scalar type of the same floating point precision as the left hand side. They return a reference to their left hand side.

The binary operators, +, -, *, /, are defined for all permutations of scalar and complex types where both operands are of the same precision and at least one is complex. They return a complex number.

The unary operators, +, -, are defined for the complex types. They return a complex number.

The equality operators, ==, !=, are defined for all permutations of scalar and complex types where both operands are of the same precision and at least one is complex. The == operator returns an integer which is nonzero if and only if its arguments are equal. The != operator returns an integer which is nonzero if and only if its arguments are not equal. Two complex numbers are equal if and only if their real components are equal and their imaginary components are equal.

6 Intrinsic functions

The intrinsic functions all their arguments by value and return either a complex number or a scalar number of the same precision as the parameters. All the intrinsics defined here are defined for all three complex classes. Details of the semantics of each operator are the same as in David Knaak's "Complex extensions to C", document X3J11/92-075.

- The decision to pass parameters by value instead of, for example, by const reference was made for two reasons. One, it matches the existing AT&T implementation. Second, it matches the method used to pass parameters to the standard C double precision math intrinsics.

The following intrinsics all take a single complex parameter and return either a complex number or a scalar as appropriate:

```
abs
norm
arg
conj
cos
cosh
exp
imag
log
real
sin
sinh
sqrt
```

There are four overloaded intrinsic pow functions for each precision:

```
complex pow(complex, int)
complex pow(complex, scalar)
complex pow(complex, complex)
complex pow(scalar, complex)
```

Here, scalar and complex refer to any of the three sets of matching floating point and complex types.

The intrinsic function `polar` takes two scalar arguments of the same precision and returns the corresponding complex number whose norm is given by the first argument and whose arg by the second argument:

```
complex polar(scalar, scalar)
```

7 Stream input/output

Input and output using the C++ iostreams classes will be provided using the standard lshift (<<) operator for output and the rshift (>>) operator for input. The operators are declared as follows (`complex` can stand for any of the three complex types)

```
ostream& operator<<(ostream&, complex)
istream& operator>>(istream&, complex&)
```

The format of complex numbers written to streams is `(re,im)` where `re` and `im`, the real and imaginary parts, are formatted in the same way as the corresponding scalar types. The parentheses are part of the format. The formatting of the components is sensitive to manipulators in the same way as the underlying scalar types.

- Perhaps the formatting should be more flexible. Is there a locale in which complex numbers are written in a different way?

Complex numbers being read from an input stream are expected to be in the format `(re,im)` where `re` and `im`, the real and imaginary parts, have the same allowed formats as the corresponding scalar types. The parentheses are part of the format.

- Perhaps the input operator should also be able to interpret scalars (i.e. as complex numbers with imaginary component zero).

Code which uses complex numbers but not iostreams should not be forced to include either the iostreams header file or link in any iostreams code.

- This implies that implementations should not put a "`#include <iostream.h>`" in `complex.h` and also should put definitions of the input and output operators in a separate compilation unit from other complex number functions.

8 Sample definitions

Here are sample definitions for a class based complex implementation:

8.1 float_complex definition

```
class float_complex {
```

```
public:
    float_complex()
    float_complex(float re);
    float_complex(float re, float im);
    float_complex(const float_complex&);
    float_complex& operator=(const float_complex&);

    float_complex& operator+=(float_complex);
    float_complex& operator-=(float_complex);
    float_complex& operator*=(float_complex);
    float_complex& operator/=(float_complex);
};

float_complex _float_complex(const double_complex&);
float_complex _float_complex(const long_double_complex&);

float_complex operator+(float,float_complex);
float_complex operator+(float_complex,float);
float_complex operator+(float_complex,float_complex);
float_complex operator-(float,float_complex);
float_complex operator-(float_complex,float);
float_complex operator-(float_complex,float_complex);
float_complex operator*(float,float_complex);
float_complex operator*(float_complex,float);
float_complex operator*(float_complex,float_complex);
float_complex operator/(float,float_complex);
float_complex operator/(float_complex,float);
float_complex operator/(float_complex,float_complex);

float_complex operator+(float_complex);
float_complex operator-(float_complex);

int operator==(float,float_complex);
int operator==(float_complex,float);
int operator==(float_complex,float_complex);
int operator!=(float,float_complex);
int operator!=(float_complex,float);
int operator!=(float_complex,float_complex);

istream& operator>>(istream&,float_complex&);
ostream& operator<<(ostream&,float_complex);

float abs(float_complex);
float norm(float_complex);
float arg(float_complex);
float real(float_complex);
float imag(float_complex);
float_complex conj(float_complex);
float_complex cos(float_complex);
float_complex cosh(float_complex);
float_complex exp(float_complex);
float_complex log(float_complex);
float_complex sin(float_complex);
float_complex sinh(float_complex);
float_complex sqrt(float_complex);

float_complex pow(float_complex,int);
float_complex pow(float_complex,float);
float_complex pow(float_complex,float_complex);
float_complex pow(float,float_complex);

float_complex polar(float,float);
```

8.2 double_complex definition

```
class double_complex {
public:
    double_complex()
    double_complex(double re);
    double_complex(double re, double im);
    double_complex(const float_complex&);
    double_complex(const double_complex&);
    double_complex& operator=(const double_complex&);

    double_complex& operator+=(double_complex);
    double_complex& operator-=(double_complex);
    double_complex& operator*=(double_complex);
    double_complex& operator/=(double_complex);
};

double_complex _double_complex(const long_double_complex&);

double_complex operator+(double,double_complex);
double_complex operator+(double_complex,double);
double_complex operator+(double_complex,double_complex);
double_complex operator-(double,double_complex);
double_complex operator-(double_complex,double);
double_complex operator-(double_complex,double_complex);
double_complex operator*(double,double_complex);
double_complex operator*(double_complex,double);
double_complex operator*(double_complex,double_complex);
double_complex operator/(double,double_complex);
double_complex operator/(double_complex,double);
double_complex operator/(double_complex,double_complex);

double_complex operator+(double_complex);
double_complex operator-(double_complex);

int operator==(double,double_complex);
int operator==(double_complex,double);
int operator==(double_complex,double_complex);
int operator!=(double,double_complex);
int operator!=(double_complex,double);
int operator!=(double_complex,double_complex);

istream& operator>>(istream&,double_complex&);
ostream& operator<<(ostream&,double_complex);

double abs(double_complex);
double norm(double_complex);
double arg(double_complex);
double real(double_complex);
double imag(double_complex);
double_complex conj(double_complex);
double_complex cos(double_complex);
double_complex cosh(double_complex);
double_complex exp(double_complex);
double_complex log(double_complex);
double_complex sin(double_complex);
double_complex sinh(double_complex);
double_complex sqrt(double_complex);

double_complex pow(double_complex,int);
double_complex pow(double_complex,double);
double_complex pow(double_complex,double_complex);
double_complex pow(double,double_complex);
```

```
double_complex polar(double,double);
```

8.3 long_double_complex definition

```
class long_double_complex {
public:
    long_double_complex()
    long_double_complex(long double re);
    long_double_complex(long double re, long double im);
    long_double_complex(const float_complex&);
    long_double_complex(const double_complex&);
    long_double_complex(const long_double_complex&);
    long_double_complex& operator=(const long_double_complex&);

    long_double_complex& operator+=(long_double_complex);
    long_double_complex& operator-=(long_double_complex);
    long_double_complex& operator*=(long_double_complex);
    long_double_complex& operator/=(long_double_complex);
};

long_double_complex operator+(long double,long_double_complex);
long_double_complex operator+(long_double_complex,long double);
long_double_complex
operator+(long_double_complex,long_double_complex);
long_double_complex operator-(long double,long_double_complex);
long_double_complex operator-(long_double_complex,long double);
long_double_complex operator-
(long_double_complex,long_double_complex);
long_double_complex operator*(long double,long_double_complex);
long_double_complex operator*(long_double_complex,long double);
long_double_complex
operator*(long_double_complex,long_double_complex);
long_double_complex operator/(long double,long_double_complex);
long_double_complex operator/(long_double_complex,long double);
long_double_complex
operator/(long_double_complex,long_double_complex);

long_double_complex operator+(long_double_complex);
long_double_complex operator-(long_double_complex);

int operator==(long double,long_double_complex);
int operator==(long_double_complex,long double);
int operator==(long_double_complex,long_double_complex);
int operator!=(long double,long_double_complex);
int operator!=(long_double_complex,long double);
int operator!=(long_double_complex,long_double_complex);

istream& operator>>(istream&,long_double_complex&);
ostream& operator<<(ostream&,long_double_complex);

long double abs(long_double_complex);
long double norm(long_double_complex);
long double arg(long_double_complex);
long double real(long_double_complex);
long double imag(long_double_complex);
long_double_complex conj(long_double_complex);
long_double_complex cos(long_double_complex);
long_double_complex cosh(long_double_complex);
long_double_complex exp(long_double_complex);
long_double_complex log(long_double_complex);
long_double_complex sin(long_double_complex);
long_double_complex sinh(long_double_complex);
long_double_complex sqrt(long_double_complex);
```

```
long_double_complex pow(long_double_complex,int);
long_double_complex pow(long_double_complex,long double);
long_double_complex pow(long_double_complex,long_double_complex);
long_double_complex pow(long double,long_double_complex);

long_double_complex polar(long double,long double);
```


223