# Complex Arithmetic in C++

*Leonie V. Rose*

*Bjarne Stroustrup*

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

This memo describes a data type `complex` providing the basic facilities for using complex arithmetic in C++. The usual arithmetic operators can be used on complex numbers and a library of standard complex mathematical functions is provided. For example:

```
#include <complex.h>

main(){
        complex xx;
        complex yy = complex(1,2.718);
        xx = log(yy/3);
        cout<<1+xx;
}
```

initializes yy as a complex number of the form `(real+imag*i)`, evaluates the expressions and prints the result: `(0.706107,1.10715)`.

The data type `complex` is implemented as a class using the data abstraction facilities in C++[1]. The arithmetic operators `+ - * /`, the assignment operators `= += -= *= /=`, and the comparison operators `== !=` are provided for complex numbers. So are the trigonometric and mathematical functions: `sin()`, `cos()`, `cosh()`, `sinh()`, `sqrt()`, `log()`, `exp()`, `conj()`, `arg()`, `abs()`, `norm()`, `pow()`. Expressions such as `(xx+1)*log(yy*log(3.2))` that involves a mixture of real and complex numbers are handled correctly. The simplest complex operations, for example `+` and `+=`, are implemented without function call overhead.

[1] Bjarne Stroustrup: "The C++ Programming Language - Reference Manual" In this volume.

**Introduction**

The C++ language does not have a built-in data type for complex numbers, but it does provide language facilities for defining new data types (see also references 2 and 3). The type `complex` was designed as a useful demonstration of the power of these facilities.

There are three plausible ways to support complex numbers in C++. First, the type `complex` could be directly supported by the compiler in the same way as the types `int` and `float` are. Alternatively, a preprocessor could be written to translate all use of complex numbers into standard C++. A third approach was used to implement type `complex`; it was specified as a user-defined type. This demonstrates that one can achieve the elegance and most of the efficiency of a built in data type without modifying the compiler. It is even much easier to implement than the pre-processor approach, which is likely to provide an inferior user interface.

This facility for complex arithmetic provides the arithmetic operators `+ / * -`, the assignment operators `= += -= *= /=`, and the comparison operators `== !=` for complex numbers. Input and output can be done using the operators `<<` ("put to") and `>>` ("get from"). The initialization functions and operator `>>` accept a Cartesian representation of a `complex`. The functions `real()` and `imag()` return the real and imaginary part of a `complex`, respectively, and operator `<<` prints a `complex` as `(real,imaginary)`. The internal representation of a `complex`, is, however, inaccessible and in principle unknown to a user. Polar coordinates can also be used. The function `polar()` creates a `complex` given its polar representation, and `abs()` and `arg()` return the polar magnitude and angle, respectively, of a `complex`. The function `norm()` returns the square of the magnitude of a `complex`. The following complex functions are also provided: `sqrt()`, `exp()`, `log()`, `sin()`, `cos()`, `sinh()`, `cosh()`, `pow()`, `conj()`. The declaration of `complex` and the declarations of the complex functions can be found in Appendix A. A complete program using complex numbers can be found in Appendix B.

**Complex Variables and Data Initialization**

A program using complex arithmetic will contain declarations of `complex` variables. For example:

```
complex zz = complex(3,-5);
```

will declare `zz` to be complex and initialize it with a pair of values. The first value of the pair is taken as the real part of the Cartesian representation of a complex number and the second as the imaginary part. The function `complex()` constructs a complex value given suitable arguments†. It is responsible for initializing `complex` variables, and will convert the arguments to the proper type `(double)`. Such initializations may be written more compactly. For example:

```
complex zz(3,-5);
complex c_name(-3.9,7);
complex rpr(SQRT_2,root3);
```

A complex variable can be initialized to a real value by using the constructor with only one argument. For example:

```
complex ra = complex(1);
```

will set up `ra` as a complex variable initialized to `(1,0)`. Alternatively the initialization to a real value can also be written without explicit use of the constructor:

```
complex rb = 123;
```

The integer value will be converted to the equivalent complex value exactly as if the constructor `complex(123)` had been used explicitly. However, no conversion of a `complex` into a `double` is defined, so

---

[2] Bjarne Stroustrup: "Data Abstraction in C++" In this volume.

[3] Bjarne Stroustrup: "Operator Overloading in C++" In this volume.

† Such a function is called a constructor. A constructor for a type always has the same name as the type itself.

```
double dd = complex(1,0);
```

is illegal and will cause a compile time error.

If there is no initialization in the declaration of a complex variable, then the variable is initialized to `(0,0)`. For example:

```
complex orig;
```

is equivalent to the declaration:

```
complex orig = complex(0,0);
```

Naturally a complex variable can also be initialized by a complex expression. For example:

```
complex cx(-0.5000000e+02,0.8660254e+02);
complex cy = cx+log(cx);
```

It is also possible to declare arrays of complex numbers. For example:

```
complex carray[30];
```

sets up an array of 30 complex numbers, all initialized to `(0,0)`. Using the above declarations:

```
complex carr[] = { cx, cy, carray[2], complex(1.1,2.2) };
```

sets up a complex array `carr[]` of four `complex` elements and initializes it with the members of the list. However, a struct style initialization cannot be used. For example:

```
complex cwrong[] = {1.5, 3.3, 4.2, 4};
```

is illegal, because it makes unwarranted assumptions about the representation of complex numbers.

**Input and Output**

Simple input and output can be done using the operators `>>` ("get from") and `<<` ("put to") . They are declared like this using the facility for overloading operators:

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

When `zz` is a complex variable `cin>>zz` reads a pair of numbers from the (standard) input stream `cin` into `zz`. The first number of the pair is interpreted as the real part of the Cartesian representation of a complex number and the second as the imaginary part. The expression `cout<<zz` writes `zz` to the (standard) output stream `cout`. For example:

```
void copy(istream& from, ostream& to)
{
        complex zz;
        while ( from>>zz ) to<<zz;
}
```

reads a stream of complex numbers like `(3.400000,5.000000)` and writes them like `(3.4,5)`. The parentheses and comma are mandatory delimiters for input, while white space is optional. A single real number, for example `10e-7` or `(123)`, will be interpreted as a `complex` with `0` as the imaginary part by operator `>>`.

A user who does not like the standard implementation of `<<` and `>>` can provide alternate versions.

**Cartesian and Polar Coordinates**

The functions `real()` and `imag()` return the real and imaginary parts of a complex number, respectively. This can, for example, be used to create differently formatted output of a `complex`:

```
complex cc = complex(3.4,5);
cout<<real(cc)<<"+"<<imag(cc)"*i";
```

will print `3.4+5*i`.

The function `polar()` creates a `complex` given a pair of polar coordinates (magnitude, angle). The functions `arg()` and `abs()` both take a `complex` argument and return the angle and magnitude (modulus), respectively. For example:

```
complex cc = polar(SQRT_2,PI/4);    /* also known as complex(1,1) */
double magn = abs(cc);              /* magn = sqrt(2) */
double angl = arg(cc);              /* angl = PI/4 */
cout<<"(m="<<magn<<", a="<<angl<<")";
```

If input and output functions for the polar representation of complex numbers are needed they can easily be written by the user.

**Arithmetic operators**

The basic arithmetic operators `+ -` (unary and binary) `/ *`, the assignment operators `= += -= *= /=`, as well as the equality operators `== !=` can be used for complex numbers. The operators have their conventional precedences. For example: `a=b*c+d` for complex variables `a`, `b`, `c`, and `d` is equivalent to `a=(b*c)+d`. There are no operators for exponentiation and conjugation; instead the functions `pow()` and `conj()` are provided. The operators `+= -= *= /=` do not produce a value that can be used in an expression; thus the following examples will cause compile time errors:

```
complex a, b;
...
if ( (a+=2)==0 ) { ... }
b = a *= b;
```

**Mixed Mode Arithmetic**

Mixed mode expressions are handled correctly. Real values will be converted to complex where necessary. For example:

```
complex xx(3.5,4.0);
complex yy = log(yy) + log(3.2);
```

This expression involves a mixture of real values: `log(3.2)`, and complex values: `log(yy)` and the sum. Another example of mixing real and complex, `xx=1` is equivalent to `xx=complex(1)` which in turn is equivalent to `xx=complex(1,0)`. The interpretation of the expression `(xx+1)*yy*3.2` is `(((xx+complex(1))*yy)*complex(3.2))`

**Mathematical Functions**

A library of complex mathematical functions is provided. A complex function typically has a counterpart of the same name in the standard mathematical library. In this case the function name will be overloaded. That is, when called, the function to be invoked will be chosen based on the argument type. For example, `log(1)` will invoke the real `log()`, and `log(complex(1))` will invoke the complex `log()`. In each case the integer `1` is converted to the real value `1.0`.

These functions will produce a result for every possible argument. If it is not possible to produce a mathematically acceptable result, the function `complex_error()` will be called and some suitable value returned. In particular, the functions try to avoid actual overflow, calling `complex_error()` with an overflow message instead. The user can supply `complex_error()`. Otherwise a function that simply sets the integer `errno` is used. See appendix C for details.

```
complex conj(complex);
```

`Conj(zz)` returns the complex conjugate of `zz`.

```
double norm(complex);
```

`Norm(zz)` returns the square of the magnitude of `zz`. It is faster than `abs(zz)`, but more likely to cause an overflow error. It is intented for comparisons of magnitudes.

```
overload pow;
double   pow(double, double);
complex  pow(double, complex);
complex  pow(complex, int);
complex  pow(complex, double);
complex  pow(complex, complex);
```

`Pow(aa,bb)` raises `aa` to the power of `bb`. For example, to calculate `(1-i)**4`:

```
cout<<pow( complex(1,-1), 4);
```

The output is `(-4,0)`.

```
overload log;
double   log(double);
complex  log(complex);
```

`Log(zz)` computes the natural logarithm of `zz`. `Log(0)` causes an error, and a huge value is returned.

```
overload exp;
double   exp(double);
complex  exp(complex);
```

`Exp(zz)` computes `e**zz`, `e` being 2.718281828...

```
overload sqrt;
double   sqrt(double);
complex  sqrt(complex);
```

`Sqrt(zz)` calculates the square root of `zz`. The trigonometric functions available are:

```
overload sin;
double   sin(double);
complex  sin(complex);

overload cos;
double   cos(double);
complex  cos(complex);
```

Hyperbolic functions are also available:

```
overload sinh;
double   sinh(double);
complex  sinh(complex);

overload cosh;
double   cosh(double);
complex  cosh(complex);
```

Other trigonometric and hyperbolic functions, for example `tan()` and `tanh()`, can be written by the user using overloaded function names.

**Efficiency**

C++'s facility for overloading function names allows `complex` to handle overloaded function calls in an efficient manner. If a function name is declared to be overloaded, and that name is invoked in a function call, then the declaration list for that function is scanned in order, and the first occurrence of the appropriate function with matching arguments will be invoked. For further detail see reference 4. For example, consider the exponential function:

```
overload exp;
double   exp(double);
complex  exp(complex);
```

When called with a `double` argument the first, and in this case most efficient, `exp()` will be

invoked. If a `complex` result is needed, the `double` result is then implicitly converted using the appropriate constructor. For example:

```
complex foo = exp(3.5);
```

is evaluated as

```
complex foo = complex( exp(3.5) );
```

and not

```
complex foo = exp( complex(3.5) );
```

Constructors can also be used explicitly. For example:

```
complex add(complex a1, complex a2) /* silly way of doing a1+a2 */
{
        return complex( real(a1)+real(a2), imag(a1)+imag(a2) );
}
```

Inline functions are used to avoid function call overhead for the simplest operations, for example, `conj()`, `+`, `+=`, and the constructors (See appendix A).

**Acknowledgments**

Phil Gillis supplied us with the complex functions used for the `cxp` package. Most of the functions presented here are modified versions of those. Stu Feldman provided us with valuable advice and some functions. Doug McIlroy's constructive comments led to a major re-write. Eric Grosse suggested the FFT function in Appendix B as an example.

**Appendix A: Type complex**

This is the definition of type `complex`. It can be included as `<complex.h>`. A `friend` declaration specifies that a function may access the internal representation of a `complex`. The file `stream.h` is included to allow declaration of the stream i/o operators `<<` and `>>` for complex numbers.

```
#include <stream.h>
#include <errno.h>

overload cos;
overload cosh;
overload exp;
overload log;
overload pow;
overload sin;
overload sinh;
overload sqrt;
overload abs;

#include <math.h>

class complex {
        double  re, im;
public:
        complex(double r = 0, double i = 0) { re=r; im=i; }

        friend  double  abs(complex);
        friend  double  norm(complex);
        friend  double  arg(complex);
        friend  complex conj(complex);
        friend  complex cos(complex);
        friend  complex cosh(complex);
        friend  complex exp(complex);
        friend  double  imag(complex);
        friend  complex log(complex);
        friend  complex pow(double, complex);
        friend  complex pow(complex, int);
        friend  complex pow(complex, double);
        friend  complex pow(complex, complex);
        friend  complex polar(double, double = 0);
        friend  double  real(complex);
        friend  complex sin(complex);
        friend  complex sinh(complex);
        friend  complex sqrt(complex);

        friend  complex operator+(complex, complex);
        friend  complex operator-(complex);
        friend  complex operator-(complex, complex);
        friend  complex operator*(complex, complex);
        friend  complex operator/(complex, complex);
        friend  int     operator==(complex, complex);
        friend  int     operator!=(complex, complex);

        void operator+=(complex);
        void operator-=(complex);
        void operator*=(complex);
        void operator/=(complex);
};
```

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

inline complex operator+(complex a1, complex a2)
{
        return complex(a1.re+a2.re, a1.im+a2.im);
}

inline complex operator-(complex a1,complex a2)
{
        return complex(a1.re-a2.re, a1.im-a2.im);
}

inline complex operator-(complex a)
{
        return complex(-a.re, a.im);
}

inline complex conj(complex a)
{
        return complex(a.re, -a.im);
}

inline int operator==(complex a, complex b)
{
        return (a.re==b.re && a.im==b.im);
}

inline int operator!=(complex a, complex b)
{
        return (a.re!=b.re || a.im!=b.im);
}

inline void complex.operator+=(complex a)
{
        re += a.re;
        im += a.im;
}

inline void complex.operator-=(complex a)
{
        re -= a.re;
        im -= a.im;
}
```

**Appendix B: A FFT Function**

Transcribed from Fortran as presented in "FFT as Nested Multiplication, with a Twist" by Carl de Boor in SIAM Sci. Stat. Comput. Vol 1 No 1, March 1980.

```
#include <complex.h>

void fftstp(complex*, int, int, int, complex*);

const NEXTMX = 12;
int prime[NEXTMX] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 };

complex* fft(complex *z1, complex *z2, int n, int inzee)
/*
        Construct the discrete Fourier transform of z1 (or z2) in the
        Cooley-Tukey way, but with a twist.

        z1[before], z2[before].
        inzee==1 means input in z1; inzee==2 means input in z2
*/
{
        int before = n;
        int after = 1;
        int next = 0;
        int now;

        do {
                int np = prime[next];
                if ( (before/np)*np < before ) {
                        if (++next < NEXTMX) continue;
                        now = before;
                        before = 1;
                }
                else {
                        now = np;
                        before /= np;
                }
                if (inzee == 1)
                        fftstp(z1, after, now, before, z2);
                else
                        fftstp(z2, after, now, before, z1);
                inzee = 3 - inzee;
                after *= now;
        } while (1 < before)

        return (inzee==1) ? z1 : z2;
}
```

```
void fftstp(complex* zin, int after, int now, int before, complex* zout)
/*
        zin(after,before,now)
        zout(after,now,before)

        there are ample scope for optimization
*/
{
    double        angle = PI2/(now*after);
    complex omega = complex(cos(angle), -sin(angle));
    complex arg = 1;
    int     j;
    for (j=0; j<now; j++) {
        int ia;
        for (ia=0; ia<after; ia++) {
            int ib;
            for (ib=0; ib<before; ib++) {
                int in;
                     /* value = zin(ia,ib,now) */
                complex value = zin[ia + ib*after + (now-1)*before*after];

                for (in=now-2; 0<=in; in--) {
                 /* value = value*arg + zin(ia,ib,in) */
                    value *= arg;
                    value += zin[ia + ib*after + in*before*after];
                }
             /* zout(ia,j,ib) = value */
                zout[ia + j*after + ib*now*after] = value;
            }
            arg *= omega;
        }
    }
}
```

The main program below calls `fft()` with a sine curve as argument. The complete unedited output is presented on the next page. All but two of the numbers ought to have been zero. The very small numbers shows the roundoff errors. Since double-precision floating-point arithmetic was used these errors are smaller than the equivalent errors obtained using the published Fortran version.

```
#include <complex.h>

main()
/*
        test fft() with a sine curve
*/
{
        int i, n=26;
        complex *z1;
        complex *z2;
        complex *zout;
        extern complex* fft(complex*, complex*, int, int);

        z1 = new complex[n];
        z2 = new complex[n];

        cout<<"input: \n";
        for (i = 0; i < n ;i++) {
                z1[i] = sin(i*PI2/n);
                cout<<z1[i]<<"\n";
        }

        errno = 0;
        zout = fft(z1, z2, n, 1);
        if (errno) cout<<"Cerror "<<errno<<" occurred\n";

        cout<<"output: \n";
        for (i = 0; i < n ;i++)
                cout<<zout[i]<<"\n";
}
```

```
input:
(0, 0)
(0.239316, 0)
(0.464723, 0)
(0.663123, 0)
(0.822984, 0)
(0.935016, 0)
(0.992709, 0)
(0.992709, 0)
(0.935016, 0)
(0.822984, 0)
(0.663123, 0)
(0.464723, 0)
(0.239316, 0)
(4.35984e-17, 0)
(-0.239316, 0)
(-0.464723, 0)
(-0.663123, 0)
(-0.822984, 0)
(-0.935016, 0)
(-0.992709, 0)
(-0.992709, 0)
(-0.935016, 0)
(-0.822984, 0)
(-0.663123, 0)
(-0.464723, 0)
(-0.239316, 0)
output:
(9.56401e-17, 0)
(-3.76665e-16, -13)
(9.39828e-17, 1.11261e-17)
(6.42219e-16, -4.20613e-17)
(7.37279e-17, 2.33319e-16)
(2.85084e-16, 2.87918e-16)
(4.03134e-17, 5.1789e-17)
(2.60865e-16, 6.78794e-17)
(-5.71667e-17, -3.86348e-17)
(2.76315e-16, 2.36902e-17)
(-6.43755e-17, -3.80255e-17)
(1.95031e-16, 9.77858e-17)
(1.49087e-16, -7.57345e-17)
(3.17224e-16, 1.64294e-17)
(1.49087e-16, 7.57345e-17)
(2.7218e-16, -4.03777e-17)
(-6.43755e-17, 3.80255e-17)
(4.93805e-16, 3.36874e-17)
(-5.71667e-17, 3.86348e-17)
(7.86047e-16, -4.11068e-18)
(4.03134e-17, -5.1789e-17)
(1.60788e-15, -1.06841e-16)
(7.37279e-17, -2.33319e-16)
(5.45186e-15, 2.42719e-16)
(9.39828e-17, -1.11261e-17)
(-1.12013e-14, 13)
```

**Appendix C: Errors and Error Handling**

These are the declarations used by the error handling:

```
int errno;
int complex_error(int, double);
```

The user can supply `complex_error()`. Otherwise a function that simply sets `errno` is used. The exceptions generated are:

| | |
|---|---|
| cosh(zz): | |
| C_COSH_RE | \|zz.re\| too large. Value with correct angle and huge magnitude returned. |
| C_COSH_IM | \|zz.im\| too large. Complex(0,0) returned. |
| | |
| exp(zz): | |
| C_EXP_RE_POS | zz.im too small. Value with correct angle and huge magnitude returned. |
| C_EXP_RE_NEG | zz.re too small. Complex(0,0) returned. |
| C_EXP_IM | \|zz.im\| too large. Complex(0,0) returned. |
| | |
| log(zz): | |
| C_LOG_0 | zz==0. Value with a large real part and zero imaginary part returned. |
| | |
| sinh(zz): | |
| C_SINH_RE | \|zz.re\| too large. Value with correct angle and huge magnitude returned. |
| C_SINH_IM | \|zz.im\| too large. Complex(0,0) returned. |