A C++ Prototype for Complex C Extensions

N426, X3J11/95-027

Jim Thomas
Taligent, Inc.
10201 N. DeAnza Blvd.
Cupertino, CA 95014-2233
jim_thomas@taligent.com

Attached are copies of C++ source files:

```
complex.h
complex_types.h
complex_operators.h
complex_functions.h
```

They provide, in C++, partial implementation for complex arithmetic, as prescribed in "Complex C Extensions" (CCE), Chapter 6 of X3J11's Technical Report on Numerical C Extensions. Their purposes are

1. to facilitate quick prototype implementation of CCE on C++ systems
2. to provide a semantic reference for C implementors of CCE

Hence they provide the full functionality of built-in arithmetic types, with minor exceptions. For a complex arithmetic library for C++ (or C with requisite extensions) one might opt to give up some arguably unneeded functionality, e.g. real op= complex, to reduce the shear size of the headers.

These headers are designed for C++ systems whose minimum evaluation format is double. Straightforward modifications serve those whose minimum evaluation format is float or long double.

All the arithmetic operators are here as inlines, except for multiplication and division by complex types, which can be provided by adding implementations for:

```
double_complex       _Cmultd  (  const double_complex &,
                                 const double_complex &       );
long_double_complex  _Cmultl  (  const long_double_complex &,
                                 const long_double_complex &  );
double_complex       _Cdivd   (  const double_complex &,
                                 const double_complex &       );
long_double_complex  _Cdivl   (  const long_double_complex &,
                                 const long_double_complex &  );
double_complex       _Crdivd  (  double,
                                 const double_complex &       );
long_double_complex  _Crdivl  (  long double,
                                 const long_double_complex &  );
```

All of the complex functions are inline too, but several invoke external complex functions, e.g. `sinh` requires

```
double_complex      _Csinhd  (  const double_complex &       );
long_double_complex _Csinhl  (  const long_double_complex &  );
```

Useful references for implementing these low-level routines include W. Kahan's "Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit", Proceedings of the joint IMA/SIAM conference on The State of the Art in Numerical Analysis, Clarendon Press (1987), and Fred Tydeman's "Merging Complex and IEEE-754" (X3J11.1/92-041).

CCE goals include compatibility with IEEE real floating-point arithmetic (IEC 559) and storage/algorithm efficiency. The inline implementations in these headers are compatible with IEEE real floating-point arithmetic. They are intended to optimize for the minimal number of floating-point operations. The execution-time efficiency of programs built from these headers will depend on the compiler's efficiency in handling structs.

Of the features of C++, the headers employ constructor and operator members (for conversions), function and operator overloading, inlining, inheritance (for the imaginary unit class), and references. They to not use templates, virtual functions, or access control, among others. Note that C++ requires a large number of overloads to implement mixed mode arithmetic for all operators. Hence, translation time may be significant on C++ systems not supporting precompiled headers.

Limitations

This is work-in-progress. Testing is incomplete.

Names of parameters and local variables (in inlines) are from the user's name space.

The conditional (?:) operator, which can't be overloaded in C++, is not supported for imaginary and complex operands

These names, not specified in "Complex C Extensions", are introduced: long_double, float_real, double_real, long_double_real.

Char and unsigned integer types are not supported in mixed mode operations.

The headers include a few mildly awkward constructs to accommodate limitations in the compilers currently available to the author.

```
#ifndef __COMPLEX__
#define __COMPLEX__
/*-----------------------------------------------------------------------------------------------

FILES
                complex.h  --  complex arithmetic

                complex_types.h
                complex_operators.h
                complex_functions.h

SYNOPSIS
                struct double_imaginary
                struct float_imaginary
                struct long_double_imaginary
                struct double_complex
                struct float_complex
                struct long_double_complex

                const ... _Imaginary_I
                #define I _Imaginary_I

                Type conversions
                Unary operators:        +  -  ++  --
                Binary operators:       +  -  *  /  ==  !=
                Compound assignments:   +=  -=  *=  /=
                Functions:      real  imag  arg  conj  proj
                                sinh  sin  cosh  cos  tan  tanh
                                asinh  asin  acosh  acos  atan  atanh
                                sqrt  exp  log  pow

                The operators provide mixed mode arithmetic for all imaginary and complex types
                and supported real types (short, int, long, float, double, long double).  The
                functions are overloaded for all floating real, imaginary, and complex types.

DESCRIPTION
                These headers provide, in C++, partial implementation for complex arithmetic,
                as prescribed in "Complex C Extensions" (CCE), Chapter 6 of X3J11's Technical
                Report on Numerical C Extensions.

AUTHOR
                Jim Thomas
                Taligent, Inc.
                July 6, 1995

-----------------------------------------------------------------------------------------------*/

#ifndef __COMPLEX_TYPES__
#include <complex_types.h>
#endif

#ifndef __COMPLEX_OPERATORS__
#include <complex_operators.h>
#endif

#ifndef __COMPLEX_FUNCTIONS__
#include <complex_functions.h>
#endif

#endif
```

```
#ifndef __COMPLEX_TYPES__
#define __COMPLEX_TYPES__

// Forward declarations
struct double_imaginary;
struct float_imaginary;
struct long_double_imaginary;
struct double_complex;
struct float_complex;
struct long_double_complex;
struct _Imaginary_unit;

// Macro for declaring type conversions between real and imaginary or complex types
#define _CRealTypeConversionDeclarations(type) \
        operator short() const; operator int() const; operator long() const; operator float() const; operator double() const; operator long double() const; \
        type(short); type(int); type(long); type(float); type(double); type(long double)

//------type double_imaginary-----------------------------------------

struct double_imaginary {
                                double_imaginary() ()

        _CRealTypeConversionDeclarations(double_imaginary);

        double                  _Im;
);

//------type float_imaginary------------------------------------------

struct float_imaginary {
                                float_imaginary() ()
                                float_imaginary(const double_imaginary&);

                                operator double_imaginary() const;

        _CRealTypeConversionDeclarations(float_imaginary);

        float                   _Im;
);

//------type long_double_imaginary------------------------------------

struct long_double_imaginary {
                                long_double_imaginary() ()
                                long_double_imaginary(const double_imaginary&);
                                long_double_imaginary(const float_imaginary&);

                                operator double_imaginary() const;
                                operator float_imaginary() const;

        _CRealTypeConversionDeclarations(long_double_imaginary);

        long double             _Im;
);

//------type double_complex-------------------------------------------

struct double_complex {
                                double_complex() ()
                                double_complex(const double_imaginary&);
                                double_complex(const float_imaginary&);
                                double_complex(const long_double_imaginary&);

                                operator double_imaginary() const;
                                operator float_imaginary() const;
```

```
                                operator long_double_imaginary() const;

        _CRealTypeConversionDeclarations(double_complex);

        double                  _Re;
        double                  _Im;
};

//------type float_complex-----------------------------------------

struct float_complex {
                                float_complex() {}
                                float_complex(const double_complex&);
                                float_complex(const double_imaginary&);
                                float_complex(const float_imaginary&);
                                float_complex(const long_double_imaginary&);

                                operator double_complex() const;
                                operator double_imaginary() const;
                                operator float_imaginary() const;
                                operator long_double_imaginary() const;

        _CRealTypeConversionDeclarations(float_complex);

        float                   _Re;
        float                   _Im;
};

//------type long_double_complex-----------------------------------

struct long_double_complex {
                                long_double_complex() {}
                                long_double_complex(const double_complex&);
                                long_double_complex(const float_complex&);
                                long_double_complex(const long_double_imaginary&);
                                long_double_complex(const double_imaginary&);
                                long_double_complex(const float_imaginary&);

                                operator double_complex() const;
                                operator float_complex() const;
                                operator long_double_imaginary() const;
                                operator double_imaginary() const;
                                operator float_imaginary() const;

        _CRealTypeConversionDeclarations(long_double_complex);

        long double             _Re;
        long double             _Im;
};

#undef _CRealTypeConversionDeclarations

//------imaginary unit constant I = _Imaginary_I-------------------------------------

struct _Imaginary_unit : public double_imaginary {
};
const double _Cunit = 1.0;
#define _Imaginary_I (*(_Imaginary_unit*)&_Cunit)
#define I _Imaginary_I

//------auxiliary types--------------------------------------

//alternate names for real types, to facilitate use of macros
typedef long double long_double;
typedef float float_real;
```

```
typedef double double_real;
typedef long double long_double_real;

//------conversion constuctors and operators----------------------------------------

// real <-> imaginary,complex
#define _CRealTypeConversionInlnes(size) \
        inline size##_imaginary::operator short() const { return 0; } \
        inline size##_imaginary::operator int() const { return 0; } \
        inline size##_imaginary::operator long() const { return 0; } \
        inline size##_imaginary::operator float() const { return 0; } \
        inline size##_imaginary::operator double() const { return 0; } \
        inline size##_imaginary::operator long double() const { return 0; } \
        inline size##_imaginary::size##_imaginary(short) { _Im = 0; } \
        inline size##_imaginary::size##_imaginary(int) { _Im = 0; } \
        inline size##_imaginary::size##_imaginary(long) { _Im = 0; } \
        inline size##_imaginary::size##_imaginary(float) { _Im = 0; } \
        inline size##_imaginary::size##_imaginary(double) { _Im = 0; } \
        inline size##_imaginary::size##_imaginary(long double) { _Im = 0; } \
        inline size##_complex::operator short() const { return _Re; } \
        inline size##_complex::operator int() const { return _Re; } \
        inline size##_complex::operator long() const { return _Re; } \
        inline size##_complex::operator float() const { return _Re; } \
        inline size##_complex::operator double() const { return _Re; } \
        inline size##_complex::operator long double() const { return _Re; } \
        inline size##_complex::size##_complex(short x) { _Re = x; _Im = 0; } \
        inline size##_complex::size##_complex(int x) { _Re = x; _Im = 0; } \
        inline size##_complex::size##_complex(long x) { _Re = x; _Im = 0; } \
        inline size##_complex::size##_complex(float x) { _Re = x; _Im = 0; } \
        inline size##_complex::size##_complex(double x) { _Re = x; _Im = 0; } \
        inline size##_complex::size##_complex(long double x) { _Re = x; _Im = 0; }
_CRealTypeConversionInlnes(float)
_CRealTypeConversionInlnes(double)
_CRealTypeConversionInlnes(long_double)
#undef _CRealTypeConversionInlnes

// imaginary <-> imaginary
inline float_imaginary::float_imaginary(const double_imaginary& yi) { _Im = yi._Im; }
inline float_imaginary::operator double_imaginary() const { double_imaginary yi; yi._Im = _Im; return yi; }
inline long_double_imaginary::long_double_imaginary(const double_imaginary& yi) { _Im = yi._Im; }
inline long_double_imaginary::long_double_imaginary(const float_imaginary& yi) { _Im = yi._Im; }
inline long_double_imaginary::operator double_imaginary() const { double_imaginary yi; yi._Im = _Im; return yi; }
inline long_double_imaginary::operator float_imaginary() const { float_imaginary yi; yi._Im = _Im; return yi; }

// complex <-> imaginary
#define _CImaginaryComplexConversionInlines(size) \
        inline size##_complex::operator float_imaginary() const { float_imaginary yi; yi._Im = _Im; return yi; } \
        inline size##_complex::operator double_imaginary() const { double_imaginary yi; yi._Im = _Im; return yi; } \
        inline size##_complex::operator long_double_imaginary() const { long_double_imaginary yi; yi._Im = _Im; return yi; } \
        inline size##_complex::size##_complex(const float_imaginary& yi) { _Re = 0; _Im = yi._Im; } \
        inline size##_complex::size##_complex(const double_imaginary& yi) { _Re = 0; _Im = yi._Im; } \
        inline size##_complex::size##_complex(const long_double_imaginary& yi) { _Re = 0; _Im = yi._Im; }
_CImaginaryComplexConversionInlines(float)
_CImaginaryComplexConversionInlines(double)
_CImaginaryComplexConversionInlines(long_double)
#undef _CImaginaryComplexConversionInlines

// complex <-> complex
inline float_complex::float_complex(const double_complex& z) { _Re = z._Re; _Im = z._Im; }
inline float_complex::operator double_complex() const { double_complex z; z._Re = _Re; z._Im = _Im; return z; }
inline long_double_complex::long_double_complex(const double_complex& z) { _Re = z._Re; _Im = z._Im; }
inline long_double_complex::long_double_complex(const float_complex& z) { _Re = z._Re; _Im = z._Im; }
inline long_double_complex::operator double_complex() const { double_complex z; z._Re = _Re; z._Im = _Im; return z; }
inline long_double_complex::operator float_complex() const { float_complex z; z._Re = _Re; z._Im = _Im; return z; }
```

```
#endif
```

```
#ifndef __COMPLEX_OPERATORS__
#define __COMPLEX_OPERATORS__

#ifndef __COMPLEX_TYPES__
#include <complex_types.h>
#endif

// Function selection macro _CF, and supporting macros
// e.g. _CF(_Cdiv,double,float) evaluates to _Cdivd
#define _CFSshortXfloat d
#define _CFSshortXdouble d
#define _CFSshortXlong_double l
#define _CFSintXfloat d
#define _CFSintXdouble d
#define _CFSintXlong_double l
#define _CFSlongXfloat d
#define _CFSlongXdouble d
#define _CFSlongXlong_double l
#define _CFSfloatXfloat d
#define _CFSfloatXdouble d
#define _CFSfloatXlong_double l
#define _CFSdoubleXfloat d
#define _CFSdoubleXdouble d
#define _CFSdoubleXlong_double l
#define _CFSlong_doubleXfloat l
#define _CFSlong_doubleXdouble l
#define _CFSlong_doubleXlong_double l
#define _Cglu(a,b) a##b
#define _Cxglu(a,b) _Cglu(a,b)
#define _CF(fname,size1,size2) _Cxglu(fname,_CFS##size1##X##size2)

// Evaluation format macro _CEF, and supporting macros
// e.g. _CEF(int,float,imaginary) evaluates to double_imaginary (w/ double evaluation)
#define _CESshortXfloat double
#define _CESshortXdouble double
#define _CESshortXlong_double long_double
#define _CESintXfloat double
#define _CESintXdouble double
#define _CESintXlong_double long_double
#define _CESlongXfloat double
#define _CESlongXdouble double
#define _CESlongXlong_double long_double
#define _CESfloatXfloat double
#define _CESfloatXdouble double
#define _CESfloatXlong_double long_double
#define _CESdoubleXfloat double
#define _CESdoubleXdouble double
#define _CESdoubleXlong_double long_double
#define _CESlong_doubleXfloat long_double
#define _CESlong_doubleXdouble long_double
#define _CESlong_doubleXlong_double long_double
#define _CEF(size1,size2,kind) _Cxglu(_CES##size1##X##size2,_##kind)

//------unary + and - ----------------------------------------

#define _CUnaryPlusMinusInlines(size) \
        inline size##_imaginary operator +(const size##_imaginary& z) { return z; } \
        inline size##_imaginary operator -(const size##_imaginary& yi) { size##_imaginary vi; vi._Im = -yi._Im; return vi; } \
        inline size##_complex operator +(const size##_complex& z) { return z; } \
        inline size##_complex operator -(const size##_complex& z) { size##_complex w; w._Re = -z._Re; w._Im = -z._Im; return w; }
_CUnaryPlusMinusInlines(float)
_CUnaryPlusMinusInlines(double)
_CUnaryPlusMinusInlines(long_double)
#undef _CUnaryPlusMinusInlines
```

```
//------internal functions----------------------------------------

double_complex          _Cmultd(const double_complex&, const double_complex&);
long_double_complex     _Cmultl(const long_double_complex&, const long_double_complex&);
double_complex          _Cdivd(const double_complex&, const double_complex&);
long_double_complex     _Cdivl(const long_double_complex&, const long_double_complex&);
double_complex          _Crdivd(double, const double_complex&);
long_double_complex     _Crdivl(long double, const long_double_complex&);

//------compound assignments--------------------------------------

// imaginary op=
#define _CImaginaryCompoundAssignmentInlines(size) \
        inline size##_imaginary& operator +=(size##_imaginary& yi,long) { return yi; } \
        inline size##_imaginary& operator +=(size##_imaginary& yi,int) { return yi; } \
        inline size##_imaginary& operator +=(size##_imaginary& yi,short) { return yi; } \
        inline size##_imaginary& operator +=(size##_imaginary& yi,long double) { return yi; } \
        inline size##_imaginary& operator +=(size##_imaginary& yi,double) { return yi; } \
        inline size##_imaginary& operator +=(size##_imaginary& yi,float) { return yi; } \
        inline size##_imaginary& operator +=(size##_imaginary& yi,const double_imaginary& vi) { yi._Im += vi._Im; return yi; } \
        inline size##_imaginary& operator +=(size##_imaginary& yi,const float_imaginary& vi) { yi._Im += vi._Im; return yi; } \
        inline size##_imaginary& operator +=(size##_imaginary& yi,const long_double_imaginary& vi) { yi._Im += vi._Im; return yi; } \
        inline size##_imaginary& operator +=(size##_imaginary& yi,const double_complex& vi) { yi._Im += vi._Im; return yi; } \
        inline size##_imaginary& operator +=(size##_imaginary& yi,const float_complex& vi) { yi._Im += vi._Im; return yi; } \
        inline size##_imaginary& operator +=(size##_imaginary& yi,const long_double_complex& vi) { yi._Im += vi._Im; return yi; } \
        inline size##_imaginary& operator -=(size##_imaginary& yi,long) { return yi; } \
        inline size##_imaginary& operator -=(size##_imaginary& yi,int) { return yi; } \
        inline size##_imaginary& operator -=(size##_imaginary& yi,short) { return yi; } \
        inline size##_imaginary& operator -=(size##_imaginary& yi,long double) { return yi; } \
        inline size##_imaginary& operator -=(size##_imaginary& yi,double) { return yi; } \
        inline size##_imaginary& operator -=(size##_imaginary& yi,float) { return yi; } \
        inline size##_imaginary& operator -=(size##_imaginary& yi,const double_imaginary& vi) { yi._Im -= vi._Im; return yi; } \
        inline size##_imaginary& operator -=(size##_imaginary& yi,const float_imaginary& vi) { yi._Im -= vi._Im; return yi; } \
        inline size##_imaginary& operator -=(size##_imaginary& yi,const long_double_imaginary& vi) { yi._Im -= vi._Im; return yi; } \
        inline size##_imaginary& operator -=(size##_imaginary& yi,const double_complex& w) { yi._Im -= w._Im; return yi; } \
        inline size##_imaginary& operator -=(size##_imaginary& yi,const float_complex& w) { yi._Im -= w._Im; return yi; } \
        inline size##_imaginary& operator -=(size##_imaginary& yi,const long_double_complex& w) { yi._Im -= w._Im; return yi; } \
        inline size##_imaginary& operator *=(size##_imaginary& yi,long x) { yi._Im *= x; return yi; } \
        inline size##_imaginary& operator *=(size##_imaginary& yi,int x) { yi._Im *= x; return yi; } \
        inline size##_imaginary& operator *=(size##_imaginary& yi,short x) { yi._Im *= x; return yi; } \
        inline size##_imaginary& operator *=(size##_imaginary& yi,long double x) { yi._Im *= x; return yi; } \
        inline size##_imaginary& operator *=(size##_imaginary& yi,double x) { yi._Im *= x; return yi; } \
        inline size##_imaginary& operator *=(size##_imaginary& yi,float x) { yi._Im *= x; return yi; } \
        inline size##_imaginary& operator *=(size##_imaginary& yi,const double_imaginary& vi) { yi._Im = (yi._Im * vi._Im, 0); return yi; } \
        inline size##_imaginary& operator *=(size##_imaginary& yi,const float_imaginary& vi) { yi._Im = (yi._Im * vi._Im, 0); return yi; } \
        inline size##_imaginary& operator *=(size##_imaginary& yi,const long_double_imaginary& vi) { yi._Im = (yi._Im * vi._Im, 0); return yi; } \
        inline size##_imaginary& operator *=(size##_imaginary& yi,const double_complex& w) { yi._Im *= (yi._Im * w._Im, w._Re); return yi; } \
        inline size##_imaginary& operator *=(size##_imaginary& yi,const float_complex& w) { yi._Im *= (yi._Im * w._Im, w._Re); return yi; } \
        inline size##_imaginary& operator *=(size##_imaginary& yi,const long_double_complex& w) { yi._Im *= (yi._Im * w._Im, w._Re); return yi; } \
        inline size##_imaginary& operator /=(size##_imaginary& yi,long x) { yi._Im /= x; return yi; } \
        inline size##_imaginary& operator /=(size##_imaginary& yi,int x) { yi._Im /= x; return yi; } \
        inline size##_imaginary& operator /=(size##_imaginary& yi,short x) { yi._Im /= x; return yi; } \
        inline size##_imaginary& operator /=(size##_imaginary& yi,long double x) { yi._Im /= x; return yi; } \
        inline size##_imaginary& operator /=(size##_imaginary& yi,double x) { yi._Im /= x; return yi; } \
        inline size##_imaginary& operator /=(size##_imaginary& yi,float x) { yi._Im /= x; return yi; } \
        inline size##_imaginary& operator /=(size##_imaginary& yi,const double_imaginary& vi) { yi._Im = (yi._Im / vi._Im, 0); return yi; } \
        inline size##_imaginary& operator /=(size##_imaginary& yi,const float_imaginary& vi) { yi._Im = (yi._Im / vi._Im, 0); return yi; } \
        inline size##_imaginary& operator /=(size##_imaginary& yi,const long_double_imaginary& vi) { yi._Im = (yi._Im / vi._Im, 0); return yi; } \
        inline size##_imaginary& operator /=(size##_imaginary& yi,const double_complex& w) { yi._Im = _CF(_Crdiv,size,double)(yi._Im, w); return yi; } \
        inline size##_imaginary& operator /=(size##_imaginary& yi,const float_complex& w) { yi._Im = _CF(_Crdiv,size,float)(yi._Im, w); return yi; } \
        inline size##_imaginary& operator /=(size##_imaginary& yi,const long_double_complex& w) { yi._Im = _Crdivl(yi._Im, w); return yi; }
_CImaginaryCompoundAssignmentInlines(float)
_CImaginaryCompoundAssignmentInlines(double)
_CImaginaryCompoundAssignmentInlines(long_double)
#undef _CImaginaryCompoundAssignmentInlines
```

```
// complex op=
#define _CComplexCompoundAssignmentInlines(size) \
        inline size##_complex& operator +=(size##_complex& z, long x) { z._Re += x; return z; } \
        inline size##_complex& operator +=(size##_complex& z, int x) { z._Re += x; return z; } \
        inline size##_complex& operator +=(size##_complex& z, short x) { z._Re += x; return z; } \
        inline size##_complex& operator +=(size##_complex& z, long double x) { z._Re += x; return z; } \
        inline size##_complex& operator +=(size##_complex& z, double x) { z._Re += x; return z; } \
        inline size##_complex& operator +=(size##_complex& z, float x) { z._Re += x; return z; } \
        inline size##_complex& operator +=(size##_complex& z,const double_imaginary& yi) { z._Im += yi._Im; return z; } \
        inline size##_complex& operator +=(size##_complex& z,const float_imaginary& yi) { z._Im += yi._Im; return z; } \
        inline size##_complex& operator +=(size##_complex& z,const long_double_imaginary& yi) { z._Im += yi._Im; return z; } \
        inline size##_complex& operator +=(size##_complex& z,const double_complex& w) { z._Re += w._Re; z._Im += w._Im; return z; } \
        inline size##_complex& operator +=(size##_complex& z,const float_complex& w) { z._Re += w._Re; z._Im += w._Im; return z; } \
        inline size##_complex& operator +=(size##_complex& z,const long_double_complex& w) { z._Re += w._Re; z._Im += w._Im; return z; } \
        inline size##_complex& operator -=(size##_complex& z, long x) { z._Re -= x; return z; } \
        inline size##_complex& operator -=(size##_complex& z, int x) { z._Re -= x; return z; } \
        inline size##_complex& operator -=(size##_complex& z, short x) { z._Re -= x; return z; } \
        inline size##_complex& operator -=(size##_complex& z, long double x) { z._Re -= x; return z; } \
        inline size##_complex& operator -=(size##_complex& z, double x) { z._Re -= x; return z; } \
        inline size##_complex& operator -=(size##_complex& z, float x) { z._Re -= x; return z; } \
        inline size##_complex& operator -=(size##_complex& z,const double_imaginary& yi) { z._Im -= yi._Im; return z; } \
        inline size##_complex& operator -=(size##_complex& z,const float_imaginary& yi) { z._Im -= yi._Im; return z; } \
        inline size##_complex& operator -=(size##_complex& z,const long_double_imaginary& yi) { z._Im -= yi._Im; return z; } \
        inline size##_complex& operator -=(size##_complex& z,const double_complex& w) { z._Re -= w._Re; z._Im -= w._Im; return z; } \
        inline size##_complex& operator -=(size##_complex& z,const float_complex& w) { z._Re -= w._Re; z._Im -= w._Im; return z; } \
        inline size##_complex& operator -=(size##_complex& z,const long_double_complex& w) { z._Re -= w._Re; z._Im -= w._Im; return z; } \
        inline size##_complex& operator *=(size##_complex& z, long x) { z._Re *= x; z._Im *= x; return z; } \
        inline size##_complex& operator *=(size##_complex& z, int x) { z._Re *= x; z._Im *= x; return z; } \
        inline size##_complex& operator *=(size##_complex& z, short x) { z._Re *= x; z._Im *= x; return z; } \
        inline size##_complex& operator *=(size##_complex& z, long double x) { z._Re *= x; z._Im *= x; return z; } \
        inline size##_complex& operator *=(size##_complex& z, double x) { z._Re *= x; z._Im *= x; return z; } \
        inline size##_complex& operator *=(size##_complex& z, float x) { z._Re *= x; z._Im *= x; return z; } \
        inline size##_complex& operator *=(size##_complex& z, const double_imaginary& yi) { size t = z._Re * yi._Im; z._Re = -z._Im * yi._Im; z._Im = t; return z; } \
        inline size##_complex& operator *=(size##_complex& z, const float_imaginary& yi) { size t = z._Re * yi._Im; z._Re = -z._Im * yi._Im; z._Im = t; return z; } \
        inline size##_complex& operator *=(size##_complex& z, const long_double_imaginary& yi) { size t = z._Re * yi._Im; z._Re = -z._Im * yi._Im; z._Im = t; return z; } \
        inline size##_complex& operator *=(size##_complex& z, const double_complex& w) { z = _CF(_Cmult,size,double)(z, w); return z; } \
        inline size##_complex& operator *=(size##_complex& z, const float_complex& w) { z = _CF(_Cmult,size,float)(z, w); return z; } \
        inline size##_complex& operator *=(size##_complex& z, const long_double_complex& w) { z = _Cmultl(z, w); return z; } \
        inline size##_complex& operator /=(size##_complex& z, long x) { z._Re /= x; z._Im /= x; return z; } \
        inline size##_complex& operator /=(size##_complex& z, int x) { z._Re /= x; z._Im /= x; return z; } \
        inline size##_complex& operator /=(size##_complex& z, short x) { z._Re /= x; z._Im /= x; return z; } \
        inline size##_complex& operator /=(size##_complex& z, long double x) { z._Re /= x; z._Im /= x; return z; } \
        inline size##_complex& operator /=(size##_complex& z, double x) { z._Re /= x; z._Im /= x; return z; } \
        inline size##_complex& operator /=(size##_complex& z, float x) { z._Re /= x; z._Im /= x; return z; } \
        inline size##_complex& operator /=(size##_complex& z, const double_imaginary& yi) { size t = -z._Re / yi._Im; z._Re = z._Im / yi._Im; z._Im = t; return z; } \
        inline size##_complex& operator /=(size##_complex& z, const float_imaginary& yi) { size t = -z._Re / yi._Im; z._Re = z._Im / yi._Im; z._Im = t; return z; } \
        inline size##_complex& operator /=(size##_complex& z, const long_double_imaginary& yi) { size t = -z._Re / yi._Im; z._Re = z._Im / yi._Im; z._Im = t; return z; } \
        inline size##_complex& operator /=(size##_complex& z, const double_complex& w) { z = _CF(_Cdiv,size,double)(z, w); return z; } \
        inline size##_complex& operator /=(size##_complex& z, const float_complex& w) { z = _CF(_Cdiv,size,float)(z, w); return z; } \
        inline size##_complex& operator /=(size##_complex& z, const long_double_complex& w) { z = _Cdivl(z, w); return z; }
_CComplexCompoundAssignmentInlines(float)
_CComplexCompoundAssignmentInlines(double)
_CComplexCompoundAssignmentInlines(long_double)
#undef _CComplexCompoundAssignmentInlines

// real op=
#define _CRealTypeCompoundAssignemntInlines(type) \
        inline type& operator +=(type& x, const float_imaginary&) { return x; } \
        inline type& operator +=(type& x, const double_imaginary&) { return x; } \
        inline type& operator +=(type& x, const long_double_imaginary&) { return x; } \
        inline type& operator +=(type& x, const float_complex& z) { return x += z._Re; } \
        inline type& operator +=(type& x, const double_complex& z) { return x += z._Re; } \
        inline type& operator +=(type& x, const long_double_complex& z) { return x += z._Re; } \
        inline type& operator -=(type& x, const float_imaginary&) { return x; } \
```

```
        inline type& operator -=(type& x, const double_imaginary&) { return x; } \
        inline type& operator -=(type& x, const long_double_imaginary&) { return x; } \
        inline type& operator -=(type& x, const float_complex& z) { return x -= z._Re; } \
        inline type& operator -=(type& x, const double_complex& z) { return x -= z._Re; } \
        inline type& operator -=(type& x, const long_double_complex& z) { return x -= z._Re; } \
        inline type& operator *=(type& x, const float_imaginary&) { x = 0; return x; } \
        inline type& operator *=(type& x, const double_imaginary&) { x = 0; return x; } \
        inline type& operator *=(type& x, const long_double_imaginary&) { x = 0; return x; } \
        inline type& operator *=(type& x, const float_complex& z) { return x *= z._Re; } \
        inline type& operator *=(type& x, const double_complex& z) { return x *= z._Re; } \
        inline type& operator *=(type& x, const long_double_complex& z) { return x *= z._Re; } \
        inline type& operator /=(type& x, const float_imaginary&) { x = 0; return x; } \
        inline type& operator /=(type& x, const double_imaginary&) { x = 0; return x; } \
        inline type& operator /=(type& x, const long_double_imaginary&) { x = 0; return x; } \
        inline type& operator /=(type& x, const float_complex& z) { x = _CF(_Crdiv,type,float)(x, z); return x; } \
        inline type& operator /=(type& x, const double_complex& z) { x = _CF(_Crdiv,type,double)(x, z); return x; } \
        inline type& operator /=(type& x, const long_double_complex& z) { x = _Crdivl(x, z); return x; }
_CRealTypeCompoundAssignemntInlines(short)
_CRealTypeCompoundAssignemntInlines(int)
_CRealTypeCompoundAssignemntInlines(long)
_CRealTypeCompoundAssignemntInlines(float)
_CRealTypeCompoundAssignemntInlines(double)
_CRealTypeCompoundAssignemntInlines(long_double)
#undef _CRealTypeCompoundAssignemntInlines

//------increment, decrement (++, --) ---------------------------------------

#define _CIncrementDecrementInlines(size) \
        inline size##_imaginary operator ++(size##_imaginary& yi) { return yi; } \
        inline size##_imaginary operator ++(size##_imaginary& yi, int) { return yi; } \
        inline size##_imaginary operator --(size##_imaginary& yi) { return yi; } \
        inline size##_imaginary operator --(size##_imaginary& yi, int) { return yi; } \
        inline size##_complex operator ++(size##_complex& z) { z._Re += 1; return z; } \
        inline size##_complex operator ++(size##_complex& z, int) { size##_complex w = z; z._Re += 1; return w; } \
        inline size##_complex operator --(size##_complex& z) { z._Re -= 1; return z; } \
        inline size##_complex operator --(size##_complex& z, int) { size##_complex w = z; z._Re -= 1; return w; }
_CIncrementDecrementInlines(float)
_CIncrementDecrementInlines(double)
_CIncrementDecrementInlines(long_double)
#undef _CIncrementDecrementInlines

//------type * I------------------------------------

#define _CFloatingMulI(size) \
        inline _CEF(size,double,imaginary) operator *(size y, const _Imaginary_unit&) { _CEF(size,double,imaginary) vi; vi._Im = y; return vi; } \
        inline _CEF(size,double,imaginary) operator *(const _Imaginary_unit&, size y) { return y * I; } \
        inline _CEF(size,double,real) operator *(const size##_imaginary& yi, const _Imaginary_unit&) { return -yi._Im; } \
        inline _CEF(size,double,real) operator *(const _Imaginary_unit&, const size##_imaginary& yi) { return yi * I; } \
        inline _CEF(size,double,complex) operator *(const size##_complex& z, const _Imaginary_unit&) { _CEF(size,double,complex) w; w._Re = -z._Im; w._Im = z._Re; return w; } \
        inline _CEF(size,double,complex) operator *(const _Imaginary_unit&, const size##_complex& z) { return z * I; }
_CFloatingMulI(float)
_CFloatingMulI(double)
_CFloatingMulI(long_double)
#undef _CFloatingMulI
inline double_imaginary operator *(short y, const _Imaginary_unit&) { double_imaginary vi; vi._Im = y; return vi; }
inline double_imaginary operator *(const _Imaginary_unit&, short y) { return y * I; }
inline double_imaginary operator *(int y, const _Imaginary_unit&) { double_imaginary vi; vi._Im = y; return vi; }
inline double_imaginary operator *(const _Imaginary_unit&, int y) { return y * I; }
inline double_imaginary operator *(long y, const _Imaginary_unit&) { double_imaginary vi; vi._Im = y; return vi; }
inline double_imaginary operator *(const _Imaginary_unit&, long y) { return y * I; }

//------real op imaginary,complex-------------------------------------

#define _CRealOpImaginaryComplexInlines(rsize,size) \
        inline _CEF(rsize,size,complex) operator +(rsize y, const size##_complex& z) { _CEF(rsize,size,complex)
```

```
        inline _CEF(rsize,size,complex) operator +(rsize x, const size##_imaginary& yi) { _CEF(rsize,size,complex) z; z._Re = x; z._Im = yi._Im; return z; } \
        inline _CEF(rsize,size,complex) operator +(const size##_imaginary& yi, rsize x) { return x + yi; } \
        inline _CEF(rsize,size,complex) operator +(const size##_complex& z, rsize x) { return x + z; } \
        inline _CEF(rsize,size,complex) operator -(rsize x, const size##_imaginary& yi) { return x + (-yi); } \
        inline _CEF(rsize,size,complex) operator -(const size##_imaginary& yi, rsize x) { return yi + (-x); } \
        inline _CEF(rsize,size,complex) operator -(rsize x, const size##_complex& z) { return x + (-z); } \
        inline _CEF(rsize,size,complex) operator -(const size##_complex& z, rsize x) { return z + (-x); } \
        inline _CEF(rsize,size,imaginary) operator *(rsize x, const size##_imaginary& yi) { _CEF(rsize,size,imaginary) vi; vi._Im = x * yi._Im; return vi; } \
        inline _CEF(rsize,size,imaginary) operator *(const size##_imaginary& yi, rsize x) { return x * yi; } \
        inline _CEF(rsize,size,complex) operator *(rsize x, const size##_complex& z) { _CEF(rsize,size,complex) w; w = z; w._Re *= x; w._Im *= x; return w; } \
        inline _CEF(rsize,size,complex) operator *(const size##_complex& z, rsize x) { return x * z; } \
        inline _CEF(rsize,size,imaginary) operator /(rsize x, const size##_imaginary& yi) { _CEF(rsize,size,imaginary) vi; vi._Im = -x / yi._Im; return vi; } \
        inline _CEF(rsize,size,imaginary) operator /(const size##_imaginary& yi, rsize x) { _CEF(rsize,size,imaginary) vi; vi._Im = yi._Im / x; return vi; } \
        inline _CEF(rsize,size,complex) operator /(rsize x, const size##_complex& z) { _CEF(rsize,size,complex) w; w = _CF(_Crdiv,rsize,size)(x, _CEF(rsize,size,complex)(z)); retu
rn w; } \
        inline _CEF(rsize,size,complex) operator /(const size##_complex& z, rsize x) { _CEF(rsize,size,complex) w; w = z; w._Re /= x; w._Im /= x; return w; } \
        inline int operator ==(rsize x, const size##_imaginary& yi) { return (x == 0) && (yi._Im == 0); } \
        inline int operator ==(const size##_imaginary& yi, rsize x) { return (x == 0) && (yi._Im == 0); } \
        inline int operator ==(rsize x, const size##_complex& z) { return (x == z._Re) && (z._Im == 0); } \
        inline int operator ==(const size##_complex& z, rsize x) { return (x == z._Re) && (z._Im == 0); } \
        inline int operator !=(rsize x, const size##_imaginary& yi) { return !(x == yi); } \
        inline int operator !=(const size##_imaginary& yi, rsize x) { return !(yi == x); } \
        inline int operator !=(rsize x, const size##_complex& z) { return !(x == z); } \
        inline int operator !=(const size##_complex& z, rsize x) { return !(z == x); }

#define _CRealOpInlines(size) \
        _CRealOpImaginaryComplexInlines(int,size) \
        _CRealOpImaginaryComplexInlines(short,size) \
        _CRealOpImaginaryComplexInlines(long,size) \
        _CRealOpImaginaryComplexInlines(float,size) \
        _CRealOpImaginaryComplexInlines(double,size) \
        _CRealOpImaginaryComplexInlines(long_double,size)
_CRealOpInlines(float)
_CRealOpInlines(double)
_CRealOpInlines(long_double)
#undef _CRealOpInlines
#undef _CRealOpImaginaryComplexInlines

//------imaginary,complex op imaginary,complex----------------------------------------

#define _CImaginaryComplexOps(size1,size2) \
        inline _CEF(size1,size2,imaginary) operator +(const size1##_imaginary& yi,const size2##_imaginary& vi) { _CEF(size1,size2,imaginary) bi; bi = yi; return bi += vi; } \
        inline _CEF(size1,size2,complex) operator +(const size1##_imaginary& yi,const size2##_complex& z) { _CEF(size1,size2,complex) w; w = z; return w += yi; } \
        inline _CEF(size1,size2,complex) operator +(const size1##_complex& z,const size2##_imaginary& yi) { _CEF(size1,size2,complex) w; w = z; return w += yi; } \
        inline _CEF(size1,size2,complex) operator +(const size1##_complex& z,const size2##_complex& w) { _CEF(size1,size2,complex) c; c = z; return c += w; } \
        inline _CEF(size1,size2,imaginary) operator -(const size1##_imaginary& yi,const size2##_imaginary& vi) { return yi + (-vi); } \
        inline _CEF(size1,size2,complex) operator -(const size1##_imaginary& yi,const size2##_complex& z) { return yi + (-z); } \
        inline _CEF(size1,size2,complex) operator -(const size1##_complex& z,const size2##_imaginary& yi) { return z + (-yi); } \
        inline _CEF(size1,size2,complex) operator -(const size1##_complex& z,const size2##_complex& w) { return z + (-w); } \
        inline _CEF(size1,size2,real) operator *(const size1##_imaginary& yi,const size2##_imaginary& vi) { _CEF(size1,size2,real) x = -yi._Im; return x *= vi._Im; } \
        inline _CEF(size1,size2,complex) operator *(const size1##_imaginary& yi,const size2##_complex& z) { _CEF(size1,size2,complex) w; w = z; return w *= yi; } \
        inline _CEF(size1,size2,complex) operator *(const size1##_complex& z,const size2##_imaginary& yi) { _CEF(size1,size2,complex) w; w = z; return w *= yi; } \
        inline _CEF(size1,size2,complex) operator *(const size1##_complex& z,const size2##_complex& w) { _CEF(size1,size2,complex) c; c = z; return c *= w; } \
        inline _CEF(size1,size2,real) operator /(const size1##_imaginary& yi,const size2##_imaginary& vi) { _CEF(size1,size2,real) x = yi._Im; return x /= vi._Im; } \
        inline _CEF(size1,size2,complex) operator /(const size1##_complex& z,const size2##_imaginary& yi) { _CEF(size1,size2,complex) w; w = z; return w /= yi; } \
        inline _CEF(size1,size2,complex) operator /(const size1##_imaginary& yi,const size2##_complex& z) { _CEF(size1,size2,complex) w; w = I*_CF(_Crdiv,size1,size2)(yi._Im,z); r
eturn w; } \
        inline _CEF(size1,size2,complex) operator /(const size1##_complex& z,const size2##_complex& w) { _CEF(size1,size2,complex) c; c = z; return c /= w; } \
        inline int operator ==(const size1##_imaginary& yi,const size2##_imaginary& vi) { return yi._Im == vi._Im; } \
        inline int operator ==(const size1##_imaginary& yi,const size2##_complex& z) { return (z._Re == 0) && (yi._Im == z._Im); } \
        inline int operator ==(const size1##_complex& z,const size2##_imaginary& yi) { return (z._Re == 0) && (yi._Im == z._Im); } \
        inline int operator ==(const size1##_complex& z,const size2##_complex& w) { return (z._Re == w._Re) && (z._Im == z._Im); } \
        inline int operator !=(const size1##_imaginary& yi,const size2##_imaginary& vi) { return !(yi == vi); } \
        inline int operator !=(const size1##_imaginary& yi,const size2##_complex& z) { return !(yi == z); } \
        inline int operator !=(const size1##_complex& z,const size2##_imaginary& yi) { return !(z == yi); } \
```

```
      inline int operator !=(const size1##_complex& z,const size2##_complex& w) { return !(z == w); }
_CImaginaryComplexOps(float,float)
_CImaginaryComplexOps(float,double)
_CImaginaryComplexOps(float,long_double)
_CImaginaryComplexOps(double,float)
_CImaginaryComplexOps(double,double)
_CImaginaryComplexOps(double,long_double)
_CImaginaryComplexOps(long_double,float)
_CImaginaryComplexOps(long_double,double)
_CImaginaryComplexOps(long_double,long_double)
#undef _CImaginaryComplexOps

#endif
```

```
#ifndef __COMPLEX_FUNCTIONS__
#define __COMPLEX_FUNCTIONS__

#ifndef __COMPLEX_TYPES__
#include <complex_types.h>
#endif

#ifndef __COMPLEX_OPERATORS__
#include <complex_operators.h>
#endif

#ifndef __FP__
#include <fp.h>
#endif

// internal functions

// real
inline long double      real(long double x) { return x; }
inline double           real(double x) { return x; }
inline double           real(float x) { return x; }
inline long double      real(const long_double_imaginary&) { return 0; }
inline double           real(const double_imaginary&) { return 0; }
inline double           real(const float_imaginary&) { return 0; }
inline long double      real(const long_double_complex& z) { return z._Re; }
inline double           real(const double_complex& z) { return z._Re; }
inline double           real(const float_complex& z) { return z._Re; }

// imag
inline long double      imag(long double) { return 0; }
inline double           imag(double) { return 0; }
inline double           imag(float) { return 0; }
inline long double      imag(const long_double_imaginary& yi) { return yi._Im; }
inline double           imag(const double_imaginary& yi) { return yi._Im; }
inline double           imag(const float_imaginary& yi) { return yi._Im; }
inline long double      imag(const long_double_complex& z) { return z._Im; }
inline double           imag(const double_complex& z) { return z._Im; }
inline double           imag(const float_complex& z) { return z._Im; }

// fabs
inline long double      fabs(const long_double_imaginary& yi) { return fabs(imag(yi)); }
inline double           fabs(const double_imaginary& yi) { return fabs(imag(yi)); }
inline double           fabs(const float_imaginary& yi) { return fabs(imag(yi)); }
inline long double      fabs(const long_double_complex& z) { return hypot(real(z), imag(z)); }
inline double           fabs(const double_complex& z) { return hypot(real(z), imag(z)); }
inline double           fabs(const float_complex& z) { return hypot(real(z), imag(z)); }

// arg
extern const long double _CPIL;             // long double pi
extern const double _CPID;                  // double pi
extern const long double _CPILd2;           // long double pi/2
extern const double _CPIDd2;                // double pi/2
double _Cargd(const double_complex& z);
long double _Cargl(const long_double_complex& z);
inline long double      arg(long double x) { if (isnan(x)) return x; return signbit(x) ? _CPIL : 0; }
inline double           arg(double x) { if (isnan(x)) return x; return signbit(x) ? _CPID : 0; }
inline double           arg(float x) { if (isnan(x)) return x; return signbit(x) ? _CPID : 0; }
inline long double      arg(const long_double_imaginary& yi) { if (isnan(imag(yi))) return yi; return copysign(_CPILd2, imag(yi)); }
inline double           arg(const double_imaginary& yi) { if (isnan(imag(yi))) return yi; return copysign(_CPIDd2, imag(yi)); }
inline double           arg(const float_imaginary& yi) { if (isnan(imag(yi))) return yi; return copysign(_CPIDd2, imag(yi)); }
inline long double      arg(const long_double_complex& z) { return _Cargl(z); }
inline double           arg(const double_complex& z) { return _Cargd(z); }
inline double           arg(const float_complex& z) { return _Cargd(z); }

// conj
```

```
inline long double      conj(long double x) { return x; }
inline double           conj(double x) { return x; }
inline float            conj(float x) { return x; }
inline long_double_imaginary    conj(const long_double_imaginary& yi) { return -yi; }
inline double_imaginary         conj(const double_imaginary& yi) { return -yi; }
inline float_imaginary          conj(const float_imaginary& yi) { return -yi; }
inline long_double_complex      conj(const long_double_complex& z) { long_double_complex w; w._Re = z._Re; w._Im = -z._Im; return w; }
inline double_complex           conj(const double_complex& z) { double_complex w; w._Re = z._Re; w._Im = -z._Im; return w; }
inline float_complex            conj(const float_complex& z) { float_complex w; w._Re = z._Re; w._Im = -z._Im; return w; }

// proj
#define _Cisinf(x) (!(isfinite(x) || isnan(x)))
inline long double      proj(long double x) { return _Cisinf(x) ? INFINITY : x; }
inline double           proj(double x) { return _Cisinf(x) ? INFINITY : x; }
inline double           proj(float x) { return _Cisinf(x) ? INFINITY : x; }
inline long_double_complex      proj(const long_double_imaginary& yi) { if (_Cisinf(imag(yi))) return INFINITY + I*copysign(0.0L,imag(yi)); return yi; }
inline double_complex           proj(const double_imaginary& yi) { if (_Cisinf(imag(yi))) return INFINITY + I*copysign(0.0,imag(yi)); return yi; }
inline double_complex           proj(const float_imaginary& yi) { if (_Cisinf(imag(yi))) return INFINITY + I*copysign(0.0,imag(yi)); return yi; }
inline long_double_complex      proj(const long_double_complex& z) { if (_Cisinf(real(z)) || _Cisinf(imag(z))) return INFINITY + I*copysign(0.0L,imag(z)); return z; }
inline double_complex           proj(const double_complex& z) { if (_Cisinf(real(z)) || _Cisinf(imag(z))) return INFINITY + I*copysign(0.0,imag(z)); return z; }
inline double_complex           proj(const float_complex& z) { if (_Cisinf(real(z)) || _Cisinf(imag(z))) return INFINITY + I*copysign(0.0,imag(z)); return z; }
#undef _Cisinf

// sinh
double_complex _Csinhd(const double_complex&);
long_double_complex _Csinhl(const long_double_complex&);
inline long_double_imaginary    sinh(const long_double_imaginary& yi) { return I * sin(imag(yi)); }
inline double_imaginary         sinh(const double_imaginary& yi) { return I * sin(imag(yi)); }
inline double_imaginary         sinh(const float_imaginary& yi) { return I * sin(imag(yi)); }
inline long_double_complex      sinh(const long_double_complex& z) { return _Csinhl(z); }
inline double_complex           sinh(const double_complex& z) { return _Csinhd(z); }
inline double_complex           sinh(const float_complex& z) { return _Csinhd(z); }

// sin
inline long_double_imaginary    sin(const long_double_imaginary& yi) { return I * sinh(imag(yi)); }
inline double_imaginary         sin(const double_imaginary& yi) { return I * sinh(imag(yi)); }
inline double_imaginary         sin(const float_imaginary& yi) { return I * sinh(imag(yi)); }
inline long_double_complex      sin(const long_double_complex& z) { return -I * sinh(I*z); }
inline double_complex           sin(const double_complex& z) { return -I * sinh(I*z); }
inline double_complex           sin(const float_complex& z) { return -I * sinh(I*z); }

// cosh
double_complex _Ccoshd(const double_complex&);
long_double_complex _Ccoshl(const long_double_complex&);
inline long double              cosh(const long_double_imaginary& yi) { return cos(imag(yi)); }
inline double                   cosh(const double_imaginary& yi) { return cos(imag(yi)); }
inline double                   cosh(const float_imaginary& yi) { return cos(imag(yi)); }
inline long_double_complex      cosh(const long_double_complex& z) { return _Ccoshl(z); }
inline double_complex           cosh(const double_complex& z) { return _Ccoshd(z); }
inline double_complex           cosh(const float_complex& z) { return _Ccoshd(z); }

// cos
inline long double              cos(const long_double_imaginary& yi) { return cosh(imag(yi)); }
inline double                   cos(const double_imaginary& yi) { return cosh(imag(yi)); }
inline double                   cos(const float_imaginary& yi) { return cosh(imag(yi)); }
inline long_double_complex      cos(const long_double_complex& z) { return cosh(I*z); }
inline double_complex           cos(const double_complex& z) { return cosh(I*z); }
inline double_complex           cos(const float_complex& z) { return cosh(I*z); }

// tanh
double_complex _Ctanhd(const double_complex&);
long_double_complex _Ctanhl(const long_double_complex&);
inline long_double_imaginary    tanh(const long_double_imaginary& yi) { return I * tan(imag(yi)); }
inline double_imaginary         tanh(const double_imaginary& yi) { return I * tan(imag(yi)); }
inline double_imaginary         tanh(const float_imaginary& yi) { return I * tan(imag(yi)); }
```

```
inline long_double_complex      tanh(const long_double_complex& z) { return _Ctanhl(z); }
inline double_complex           tanh(const double_complex& z) { return _Ctanhd(z); }
inline double_complex           tanh(const float_complex& z) { return _Ctanhd(z); }

// tan
inline long_double_imaginary    tan(const long_double_imaginary& yi) { return I * tanh(imag(yi)); }
inline double_imaginary         tan(const double_imaginary& yi) { return I * tanh(imag(yi)); }
inline double_imaginary         tan(const float_imaginary& yi) { return I * tanh(imag(yi)); }
inline long_double_complex      tan(const long_double_complex& z) { return -I * tanh(I * z); }
inline double_complex           tan(const double_complex& z) { return -I * tanh(I * z); }
inline double_complex           tan(const float_complex& z) { return -I * tanh(I * z); }

// asin
double_complex _Casind(const double_complex&);
long_double_complex _Casinl(const long_double_complex&);
inline long_double_imaginary    asin(const long_double_imaginary& yi) { return I * asinh(imag(yi)); }
inline double_imaginary         asin(const double_imaginary& yi) { return I * asinh(imag(yi)); }
inline double_imaginary         asin(const float_imaginary& yi) { return I * asinh(imag(yi)); }
inline long_double_complex      asin(const long_double_complex& z) { return _Casinl(z); }
inline double_complex           asin(const double_complex& z) { return _Casind(z); }
inline double_complex           asin(const float_complex& z) { return _Casind(z); }

// asinh
inline long_double_imaginary    asinh(const long_double_imaginary& yi) { return I * asin(imag(yi)); }
inline double_imaginary         asinh(const double_imaginary& yi) { return I * asin(imag(yi)); }
inline double_imaginary         asinh(const float_imaginary& yi) { return I * asin(imag(yi)); }
inline long_double_complex      asinh(const long_double_complex& z) { return -I * asin(I*z); }
inline double_complex           asinh(const double_complex& z) { return -I * asin(I*z); }
inline double_complex           asinh(const float_complex& z) { return -I * asin(I*z); }

// acos
double_complex _Cacosd(const double_complex&);
long_double_complex _Cacosl(const long_double_complex&);
inline long_double_complex      acos(const long_double_imaginary& yi) { return _Cacosl(yi); }
inline double_complex           acos(const double_imaginary& yi) { return _Cacosd(yi); }
inline double_complex           acos(const float_imaginary& yi) { return _Cacosd(yi); }
inline long_double_complex      acos(const long_double_complex& z) { return _Cacosl(z); }
inline double_complex           acos(const double_complex& z) { return _Cacosd(z); }
inline double_complex           acos(const float_complex& z) { return _Cacosd(z); }

// acosh
double_complex _Cacoshd(const double_complex&);
long_double_complex _Cacoshl(const long_double_complex&);
inline long_double_complex      acosh(const long_double_imaginary& yi) { return _Cacoshl(yi); }
inline double_complex           acosh(const double_imaginary& yi) { return _Cacoshd(yi); }
inline double_complex           acosh(const float_imaginary& yi) { return _Cacoshd(yi); }
inline long_double_complex      acosh(const long_double_complex& z) { return _Cacoshl(z); }
inline double_complex           acosh(const double_complex& z) { return _Cacoshd(z); }
inline double_complex           acosh(const float_complex& z) { return _Cacoshd(z); }

// atanh
double_complex _Catanhd(const double_complex&);
long_double_complex _Catanhl(const long_double_complex&);
inline long_double_imaginary    atanh(const long_double_imaginary& yi) { return I * atan(imag(yi)); }
inline double_imaginary         atanh(const double_imaginary& yi) { return I * atan(imag(yi)); }
inline double_imaginary         atanh(const float_imaginary& yi) { return I * atan(imag(yi)); }
inline long_double_complex      atanh(const long_double_complex& z) { return _Catanhl(z); }
inline double_complex           atanh(const double_complex& z) { return _Catanhd(z); }
inline double_complex           atanh(const float_complex& z) { return _Catanhd(z); }

// atan
inline long_double_imaginary    atan(const long_double_imaginary& yi) { return I * atanh(imag(yi)); }
inline double_imaginary         atan(const double_imaginary& yi) { return I * atanh(imag(yi)); }
inline double_imaginary         atan(const float_imaginary& yi) { return I * atanh(imag(yi)); }
inline long_double_complex      atan(const long_double_complex& z) { return -I * atanh(I * z); }
```

```
nline double_complex          atan(const double_complex& z) { return -I * atanh(I * z); }
nline double_complex          atan(const float_complex& z) { return -I * atanh(I * z); }

/ exp
ouble_complex _Cexpd(const double_complex&);
ong_double_complex _Cexpl(const long_double_complex&);
nline long_double_complex     exp(const long_double_imaginary& yi) { return _Cexpl(yi); }
nline double_complex          exp(const double_imaginary& yi) { return _Cexpd(yi); }
nline double_complex          exp(const float_imaginary& yi) { return _Cexpd(yi); }
nline long_double_complex     exp(const long_double_complex& z) { return _Cexpl(z); }
nline double_complex          exp(const double_complex& z) { return _Cexpd(z); }
nline double_complex          exp(const float_complex& z) { return _Cexpd(z); }

/ log
nline double_complex _Clogd(const double_complex&);
nline long_double_complex _Clogl(const long_double_complex&);
nline long_double_complex     log(const long_double_imaginary& yi) { return _Clogl(yi); }
nline double_complex          log(const double_imaginary& yi) { return _Clogd(yi); }
nline double_complex          log(const float_imaginary& yi) { return _Clogd(yi); }
nline long_double_complex     log(const long_double_complex& z) { return _Clogl(z); }
nline double_complex          log(const double_complex& z) { return _Clogd(z); }
nline double_complex          log(const float_complex& z) { return _Clogd(z); }

/ sqrt
louble_complex _Csqrtd(const double_complex&);
ong_double_complex _Csqrtl(const long_double_complex&);
nline long_double_complex     sqrt(const long_double_imaginary& yi) { return _Csqrtl(yi); }
nline double_complex          sqrt(const double_imaginary& yi) { return _Csqrtd(yi); }
nline double_complex          sqrt(const float_imaginary& yi) { return _Csqrtd(yi); }
nline long_double_complex     sqrt(const long_double_complex& z) { return _Csqrtl(z); }
nline double_complex          sqrt(const double_complex& z) { return _Csqrtd(z); }
nline double_complex          sqrt(const float_complex& z) { return _Csqrtd(z); }

/ pow
louble_complex _Cpowd(const double_complex&, const double_complex&);
ong_double_complex _Cpowl(const long_double_complex&, const long_double_complex&);
define _CPowInlines(size1,size2) \
        inline _CEF(size1,size2,complex) pow(size1##_real a,const size2##_imaginary& b) { return _CF(_Cpow,size1,size2)(a,b); } \
        inline _CEF(size1,size2,complex) pow(size1##_real a,const size2##_complex& b) { return _CF(_Cpow,size1,size2)(a,b); } \
        inline _CEF(size1,size2,complex) pow(const size1##_imaginary& a,size2##_real b) { return _CF(_Cpow,size1,size2)(a,b); } \
        inline _CEF(size1,size2,complex) pow(const size1##_imaginary& a,const size2##_imaginary& b) { return _CF(_Cpow,size1,size2)(a,b); } \
        inline _CEF(size1,size2,complex) pow(const size1##_imaginary& a,const size2##_complex& b) { return _CF(_Cpow,size1,size2)(a,b); } \
        inline _CEF(size1,size2,complex) pow(const size1##_complex& a,size2##_real b) { return _CF(_Cpow,size1,size2)(a,b); } \
        inline _CEF(size1,size2,complex) pow(const size1##_complex& a,const size2##_imaginary& b) { return _CF(_Cpow,size1,size2)(a,b); } \
        inline _CEF(size1,size2,complex) pow(const size1##_complex& a,const size2##_complex& b) { return _CF(_Cpow,size1,size2)(a,b); }
_CPowInlines(float,float)
_CPowInlines(float,double)
_CPowInlines(float,long_double)
_CPowInlines(double,float)
_CPowInlines(double,double)
_CPowInlines(double,long_double)
_CPowInlines(long_double,float)
_CPowInlines(long_double,double)
_CPowInlines(long_double,long_double)
undef _CPowInlines

endif
```