C9X Complex Arithmetic

(Draft 1997-01-24)

Jim Thomas
517 Hendon Court
Sunnyvale, CA 94087
`jthomas@best.com`

Fred J. Tydeman
Tydeman Consulting
3711 Del Robles Drive
Austin, TX 78727-1814
`tydeman@tybor.com`

This specification has been accepted for inclusion in the C9X draft standard. It is WG14/N596, X3J11/96-060 (Draft 1996-09-12), but with the changes approved at the SC22/WG14 meetings in Toronto last month. These changes are

1. The changes in the *Monday paper*, “Complex Arithmetic Edits—Update”, WG14/N620 X3J11/96-84 (Draft 1996-10-15)
2. Use of the translation directive macros (**`FP_CONTRACT_`**xxx, **`CX_LIMITED_RANGE_`**xxx) no longer entails a trailing semicolon
3. Use of the translation directive macros (**`CX_LIMITED_RANGE_`**xxx) inside a compound statement is now required to be before any explicit declarations (with the word *explicit* added)
4. The style for curly braces in examples now matches the current standard.

Additional changes not discussed in Toronto are

5. The special case behavior for the **`cexp`** function for IEC 559 implementations has been changed slightly to be more consistent with the other functions.

6. The complex division sample has been fixed to handle certain infinity/zero cases.

(Note that the page number is not printed on the blank pages inserted in order for major sections to begin on right-hand pages.)

C9X Edits

6.1.1 Keywords

In Syntax, add the keyword:

`complex`

Append to Semantics this paragraph:

The token **`complex`** is a keyword in translation units where the header **`<complex.h>`** is included, but not otherwise. If the token is used prior to the first inclusion of the header, the behavior is undefined.

6.1.2.5 Types

In the first sentence of [6], replace "floating types" with "real floating types".

After [6], insert these paragraphs:

There are three *complex types*, designated as **`float complex`**, **`double complex`**, and **`long double complex`**. The real floating and complex types are collectively called the *floating types*.

For each floating type there is a *corresponding real type*, which is always a real floating type. For real floating types, it is the same type. For complex types, it is the type given by deleting the keyword **`complex`** from the type name.

Each complex type has the same representation and alignment requirements as a structured type containing exactly two elements of the corresponding real type; the first element is equal to the real part, and the second element to the imaginary part, of the complex number.

Footnote the first sentence of the first inserted paragraph above with:

Annex Y specifies pure imaginary types, for implementations supporting the infinite, NaN, and signed-zero values in IEC 559.

After the first sentence in [13], insert:

The integral and real floating types are collectively called the *real types*.

In the last sentence in [13], replace "floating types" *with* "real floating types".

After [14], insert the paragraph:

Each arithmetic type belongs to one *type-domain*. The *real type-domain* comprises the real types. The *complex type-domain* comprises the complex types.

6.2.1.3 Floating and integral

Replace all occurrences (including in the title) of "floating" *with* "real floating".

6.2.1.4 Floating types

*In the title, replace "*Floating*" with "*Real floating*".*

6.2.1 Arithmetic operands

After subclause 6.2.1.4, add these subclauses, and renumber 6.2.1.5:

6.2.1.5 Complex types

When a value of complex type is converted to another complex type, both the real and imaginary parts follow the conversion rules for the corresponding real types.

6.2.1.6 Real and complex

When a value of real type is converted to a complex type, the real part of the complex result value is determined by the rules of conversion to the corresponding real type and the imaginary part of the complex result value is a positive zero or an unsigned zero.

When a value of complex type is converted to a real type, the imaginary part of the complex value is discarded and the value of the real part is converted according to the conversion rules for the corresponding real type.

6.2.1.5 (before insertions above) Usual arithmetic conversions

Replace the second and third sentence up through the third list item with

The purpose is to determine a *common real type* for the operands and result. For the operands, each operand is converted, without change of type-domain, to a type whose corresponding real type is the common real type. The common real type is also the corresponding real type of the result, whose type-domain is determined by the operator. This pattern is called the *usual arithmetic conversions*:

> First, if the corresponding real type of either operand is **`long double`**, the other operand is converted, without change of type-domain, to a type whose corresponding real type is **`long double`**.
>
> Otherwise, if the corresponding real type of either operand is **`double`**, the other operand is converted, without change of type-domain, to a type whose corresponding real type is **`double`**.
>
> Otherwise, if the corresponding real type of either operand is **`float`**, the other operand is converted, without change of type-domain, to a type whose corresponding real type is **`float`**.

Footnote the second sentence of the inserted text above with:

For example, addition of a **`double complex`** and a **`float`** entails just the conversion of the **`float`** operand to **`double`** (and yields a **`double complex`** result).

6.3.5 Multiplicative operators

Append to Semantics the paragraph:

If either operand has complex type, then the result has complex type.

6.3.6 Additive operators

Append to Semantics the paragraph:

If either operand has complex type, then the result has complex type.

6.3.8 Relational operators

In the first bullet in [3], replace "arithmetic" *with* "real".

6.3.9 Equality operators

Append to Semantics the paragraph:

Values of complex types are equal if and only if both their real parts are equal and also their imaginary parts are equal. Any two values of arithmetic types are equal if and only if the results of their conversion to the complex type corresponding to the common real type determined by the usual arithmetic conversions are equal.

5.2.4.2.2 Characteristics of floating types `<float.h>`

In the second sentence of the paragraph defining `FLT_EVAL_METHOD` *(from FP->C9X), footnote* `FLT_EVAL_METHOD` *with:*

The evaluation method determines evaluation formats of expressions involving all floating types, not just real types. For example, if **`FLT_EVAL_METHOD`** is 1, then the product of two **`float complex`** operands is represented in the **`double complex`** format, and its parts are evaluated to **`double`**.

6.3.2.4 Postfix increment and decrement operators

In Constraints replace "scalar" *with* "real or pointer".

6.3.3.1 Prefix increment and decrement operators

In Constraints replace "scalar" *with* "real or pointer".

6.5.2 Type specifiers

In Syntax, add to the list of type specifiers:

`complex`

In Constraints, add to the bullet items:

— **`float complex`**
— **`double complex`**
— **`long double complex`**

6.5.6 Type definitions

*Rework example 1 to use something other than complex, e.g. replace "***`re`***" with "***`hi`***", "***`im`***" with "***`lo`***", and "***`complex`***" with "***`double_double`***".*

Complex Arithmetic `<complex.h>`

This section specifies a new header **`<complex.h>`**for inclusion in C9X.

7.x Complex arithmetic `<complex.h>`

The header **`<complex.h>`**defines macros and declares functions that support complex arithmetic. Each synopsis specifies a function with one or two **`double complex`** parameters and returning a **`double complex`**or **`double`** value; for each such function, there are similar functions with the same name but with **`f`** and **`l`** suffixes. The **`f`** suffix indicates that **`float`** (instead of **`double`**) is the corresponding real type for the parameters and result. Similarly the **`l`** suffix indicates that **`long double`**is the corresponding real type for the parameters and result.

The macro

```
_Imaginary_I
```

expands to an expression with a const-qualified type whose corresponding real type is the narrowest real floating type used for expression evaluation[1], and with the value of the imaginary unit[2]. The macro is suitable for use in constant expressions.[3]

The macro

```
I
```

is defined to be **`_Imaginary_I`** Notwithstanding the provisions of subclause 7.1.3, it is permitted to undefine the macro **`I`**.

7.x.1 The CX_LIMITED_RANGE macros

Synopsis

```
#include <complex.h>
CX_LIMITED_RANGE_ON
CX_LIMITED_RANGE_OFF
CX_LIMITED_RANGE_DEFAULT
```

The usual mathematical formula for multiplication of two complex numbers and the one for division by a complex number are problematic because of their treatment of infinities and because of undue overflow and underflow. The **`CX_LIMITED_RANGE`** macros can be used to inform the implementation that (where the state is *on*) the usual

---

[1] If **`FLT_EVAL_METHOD`**equals 0, 1, or 2, then **`_Imaginary_I`**has the corresponding real type **`float`**, **`double`**, or **`long double`**, respectively.

[2] The imaginary unit is a number *i* such that *i***i* = -1.

[3] For those implementations that do not support imaginary types (specified in Annex Y), the macro **`_Imaginary_I`**is intended to have a complex type.

mathematical formulas for multiplication and division are acceptable.[4] Each macro can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the macro takes effect from its occurrence until another **`CX_LIMITED_RANGE`**macro is encountered, or until the end of the translation unit. When inside a compound statement, the macro takes effect from its occurrence until another **`CX_LIMITED_RANGE`**macro is encountered (within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the macros is restored to its condition just before the compound statement. The effect of one of these macros in any other context is undefined. The default state for the macros is *off*.

7.x.2 Complex functions

Values are interpreted as radians, not degrees. An implementation may set **`errno`** but is not required to.

7.x.2.1 Branch cuts

Some of the functions below have branch cuts, across which the function is discontinuous. For implementations with a signed zero (including all IEC 559 implementations) that follow the specification of Annex Y, the sign of zero distinguishes one side of a cut from another so the function is continuous (except for format limitations) as the cut is approached from either side. For example, for the square root function, which has a branch cut along the negative real axis, the top of the cut, with imaginary part +0, maps to the positive imaginary axis, and the bottom of the cut, with imaginary part -0, maps to the negative imaginary axis.

Implementations that do not support a signed zero (see Annex X) cannot distinguish the sides of branch cuts. These implementations must map a cut so the function is continuous as the cut is approached coming around the finite endpoint of the cut in a counter clockwise direction. (Branch cuts for the functions specified here have just one finite endpoint.) For example, for the square root function, coming counter clockwise around the finite endpoint of the cut along the negative real axis approaches the cut from above, so the cut maps to the positive imaginary axis.

7.x.2.2 The `cacos` function

Synopsis

```
#include <complex.h>
double complex cacos(double complex z);
```

Description

The **`cacos`** function computes the complex arc cosine of **`z`**, with branch cuts outside the interval [-1, 1] along the real axis.

[4] The purpose of the macros is to allow the implementation to use the formulas

$(x + y*I) * (u + v*I) = (x*u - y*v) + (y*u + x*v)*I$

$(x + y*I) / (u + v*I) = (x*u + y*v) / (u*u + v*v) + ((y*u - x*v) / (u*u + v*v))*I$

where the programmer can determine they are safe.

Returns

The **`cacos`** function returns the complex arc cosine of **`z`**, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [0, π] along the real axis.

7.x.2.3 The `casin` function

Synopsis

```
#include <complex.h>
double complex casin(double complex z);
```

Description

The **`casin`** function computes the complex arc sine of **`z`**, with branch cuts outside the interval [-1, 1] along the real axis.

Returns

The **`casin`** function returns the complex arc sine of **`z`**, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [-π/2, π/2] along the real axis.

7.x.2.4 The `catan` function

Synopsis

```
#include <complex.h>
double complex catan(double complex z);
```

Description

The **`catan`** function computes the complex arc tangent of **`z`**, with branch cuts outside the interval [-i, i] along the imaginary axis.

Returns

The **`catan`** function returns the complex arc tangent of **`z`**, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [-π/2, π/2] along the real axis.

7.x.2.5 The `ccos` function

Synopsis

```
#include <complex.h>
double complex ccos(double complex z);
```

Description

The **`ccos`** function computes the complex cosine of **`z`**.

Returns

The `ccos` function returns the complex cosine of `z`.

7.x.2.6 The `csin` function

Synopsis

```
#include <complex.h>
double complex csin(double complex z);
```

Description

The `csin` function computes the complex sine of `z`.

Returns

The `csin` function returns the complex sine of `z`.

7.x.2.7 The `ctan` function

Synopsis

```
#include <complex.h>
double complex ctan(double complex z);
```

Description

The `ctan` function computes the complex tangent of `z`.

Returns

The `ctan` function returns the complex tangent of `z`.

7.x.2.8 The `cacosh` function

Synopsis

```
#include <complex.h>
double complex cacosh(double complex z);
```

Description

The `cacosh` function computes the complex arc hyperbolic cosine of `z`, with a branch cut at values less than 1 along the real axis.

Returns

The `cacosh` function returns the complex arc hyperbolic cosine of `z`, in the range of a half-strip of non-negative values along the real axis and in the interval [$-i\pi$, $i\pi$] along the imaginary axis.

7.x.2.9 The `casinh` function

Synopsis

```
#include <complex.h>
double complex casinh(double complex z);
```

Description

The `casinh` function computes the complex arc hyperbolic sine of `z`, with branch cuts outside the interval [-*i*, *i*] along the imaginary axis.

Returns

The `casinh` function returns the complex arc hyperbolic sine of `z`, in the range of a strip mathematically unbounded along the real axis and in the interval [-*iπ*/2, *iπ*/2] along the imaginary axis.

7.x.2.10 The `catanh` function

Synopsis

```
#include <complex.h>
double complex catanh(double complex z);
```

Description

The `catanh` function computes the complex arc hyperbolic tangent of `z`, with branch cuts outside the interval [-1, 1] along the real axis.

Returns

The `catanh` function returns the complex arc hyperbolic tangent of `z`, in the range of a strip mathematically unbounded along the real axis and in the interval [-*iπ*/2, *iπ*/2] along the imaginary axis.

7.x.2.11 The `ccosh` function

Synopsis

```
#include <complex.h>
double complex ccosh(double complex z);
```

Description

The `ccosh` function computes the complex hyperbolic cosine of `z`.

Returns

The `ccosh` function returns the complex hyperbolic cosine of `z`.

7.x.2.12 The `csinh` function

Synopsis

```
#include <complex.h>
double complex csinh(double complex z);
```

Description

The `csinh` function computes the complex hyperbolic sine of `z`.

Returns

The `csinh` function returns the complex hyperbolic sine of `z`.

7.x.2.13 The `ctanh` function

Synopsis

```
#include <complex.h>
double complex ctanh(double complex z);
```

Description

The `ctanh` function computes the complex hyperbolic tangent of `z`.

Returns

The `ctanh` function returns the complex hyperbolic tangent of `z`.

7.x.2.14 The `cexp` function

Synopsis

```
#include <complex.h>
double complex cexp(double complex z);
```

Description

The `cexp` function computes the complex base-e exponential of `z`.

Returns

The `cexp` function returns the complex base-e exponential of `z`.

7.x.2.15 The `clog` function

Synopsis

```
#include <complex.h>
double complex clog(double complex z);
```

Description

The `clog` function computes the complex natural (base-e) logarithm of `z`, with a branch cut along the negative real axis.

Returns

The `clog` function returns the complex natural logarithm of `z`, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi, i\pi]$ along the imaginary axis.

7.x.2.16 The `csqrt` function

Synopsis

```
#include <complex.h>
double complex csqrt(double complex z);
```

Description

The `csqrt` function computes the complex square root of `z`, with a branch cut along the negative real axis.

Returns

The `csqrt` function returns the complex square root of `z`, in the range of the right half-plane (including the imaginary axis).

7.x.2.17 The `cabs` function

Synopsis

```
#include <complex.h>
double cabs(double complex z);
```

Description

The `cabs` function computes the complex absolute value (also called norm, modulus, or magnitude) of `z`.

Returns

The `cabs` function returns the complex absolute value of `z`.

7.x.2.18 The `cpow` function

Synopsis

```
#include <complex.h>
double complex cpow(double complex x, double complex y);
```

Description

The **`cpow`** function computes the complex power function **`x`**$^{\mathtt{y}}$, with a branch cut for the first parameter along the negative real axis.

Returns

The **`cpow`** function returns the complex power function **`x`**$^{\mathtt{y}}$.

7.x.2.19 The `carg` function

Synopsis

```
#include <complex.h>
double carg(double complex z);
```

Description

The **`carg`** function computes the argument or phase angle of **`z`**, with a branch cut along the negative real axis.

Returns

The **`carg`** function returns the argument or phase angle of **`z`**, in the range $[-\pi, \pi]$.

7.x.2.20 The `conj` function

Synopsis

```
#include <complex.h>
double complex conj(double complex z);
```

Description

The **`conj`** function computes the complex conjugate of **`z`**, by reversing the sign of its imaginary part.

Returns

The **`conj`** function returns the complex conjugate of **`z`**.

7.x.2.21 The `cimag` function

Synopsis

```
#include <complex.h>
double cimag(double complex z);
```

Description

The **`cimag`** function computes the imaginary part of **`z`**.[5]

[5] For a variable **`z`** of complex type, **`z == creal(z) + cimag(z)*I`**

Returns

The `cimag` function returns the imaginary part of `z` (as a real).

7.x.2.22 The `cproj` function

Synopsis

```
    #include <complex.h>
    double complex cproj(double complex z);
```

Description

The `cproj` function computes a projection of `z` onto the Riemann sphere: `z` projects to `z` except that all complex infinities (even ones with one infinite part and one NaN part) project to positive infinity on the real axis. If `z` has an infinite part, then `cproj(z)` is equivalent to `INFINITY + I * copysign(0.0, cimag(z))`.

Returns

The `cproj` function returns a projection of `z` onto the Riemann sphere.

7.x.2.23 The `creal` function

Synopsis

```
    #include <complex.h>
    double creal(double complex z);
```

Description

The `creal` function computes the real part of `z`.

Returns

The `creal` function returns the real part of `z`.

Annex Y: IEC 559-Compatible Complex Arithmetic

Insert the following annex:

Annex Y
(informative)
IEC 559-compatible complex arithmetic

Y.1 Introduction

This annex supplements Annex X to specify complex arithmetic for compatibility with IEC 559 real floating-point arithmetic. An implementation supports this specification if and only if it defines the macro **`__STDC_IEC_559_COMPLEX__`**; the macro expands to the decimal constant 1.

Y.2 Keywords

The syntax in subclause 6.1.1 is extended to include the keyword

`imaginary`

The token **`imaginary`** is a keyword in translation units where the header **`<complex.h>`** is included, but not otherwise. If the token is used prior to the first inclusion of the header, the behavior is undefined.

Y.3 Types

There are three *imaginary types*, designated as **`float imaginary`**, **`double imaginary`**, and **`long double imaginary`**. The imaginary types (along with the real floating and complex types) are floating types.

For imaginary types, the corresponding real type is given by deleting the keyword **`imaginary`** from the type name.

Each imaginary type has the same representation and alignment requirements as the corresponding real type. The value of an object of imaginary type is the value of the real representation times the imaginary unit.

The *imaginary type-domain* comprises the imaginary types.

Y.4 Conversions

Y.4.1 Imaginary types

Conversions among imaginary types follow rules analogous to those for real floating types.

Y.4.2 Real and imaginary

When a value of imaginary type is converted to a real type, the result is a positive zero.

When a value of real type is converted to an imaginary type, the result is a positive imaginary zero.

Y.4.3 Imaginary and complex

When a value of imaginary type is converted to a complex type, the real part of the complex result value is a positive zero and the imaginary part of the complex result value is determined by the conversion rules for the corresponding real types.

When a value of complex type is converted to an imaginary type, the real part of the complex value is discarded and the value of the imaginary part is converted according to the conversion rules for the corresponding real types.

Y.5 Binary operators

The following subclauses supplement 6.3 in order to specify the type of the result for an operation with an imaginary operand.

For most operand types, the value of the result of a binary operator with an imaginary or complex operand is completely determined, with reference to real arithmetic, by the usual mathematical formula. For some operand types, the usual mathematical formula is problematic because of its treatment of infinities and because of undue overflow or underflow (7.X.1); in these cases the result satisfies certain properties (specified in Y.5.1), but is not completely determined.

Y.5.1 Multiplicative operators

Semantics

If one operand has real type and the other operand has imaginary type, then the result has imaginary type. If both operands have imaginary type, then the result has complex type. (If either operand has complex type, then the result has complex type.)

If the operands are not both complex, then the result and exception behavior of the `*` operator is defined by the usual mathematical formula:

*	real x	imaginary $y*I$	complex $x + y*I$
real u	$x*u$	$(y*u)*I$	$(x*u) + (y*u)*I$
imaginary $v*I$	$(x*v)*I$	$-y*v + 0*I$	$(-y*v) + (x*v)*I$
complex $u + v*I$	$(x*u) + (x*v)*I$	$(-y*v) + (y*u)*I$	

If the second operand is not complex, then the result of the `/` operator is defined by the usual mathematical formula:

/	*x*	*y*I*	*x + y*I*
u	*x/u*	*(y/u)*I*	*(x/u) + (y/u)*I*
*v*I*	*(-x/v)*I*	*y/v + 0*I*	*(y/v) + (-x/v)*I*

A complex or imaginary value with at least one infinite part is regarded as an *infinity* (even if its other part is a NaN). A complex or imaginary value is a *finite number* if each of its parts is a finite number (neither infinite nor NaN). A complex or imaginary value is a *zero* if each of its parts is a zero. The **`*`** and **`/`** operators satisfy the following infinity properties for all real, imaginary, and complex operands[6]:

- If one operand is an infinity and the other operand is a nonzero finite number or an infinity, then the result of the **`*`** operator is an infinity.
- If the first operand is an infinity and the second operand is a finite number, then the result of the **`/`** operator is an infinity.
- If the first operand is a finite number and the second operand is an infinity, then the result of the **`/`** operator is a zero.
- If the first operand is a nonzero finite number or an infinity and the second operand is a zero, then the result of the **`/`** operator is an infinity.

If both operands of the **`*`** operator are complex or if the second operand of the **`/`** operator is complex, the operator raises exceptions if appropriate for the calculation of the parts of the result, and may raise spurious exceptions.

Examples

1. Multiplication of **`double complex`** operands could be implemented as follows. Note that the imaginary unit **`I`** has imaginary type (see Y.6).

```
#include <math.h>
#include <complex.h>
#define isinf(x) (fabs(x)==INFINITY)

/* Multiply z * w ... */
double complex _Cmultd(double complex z, double complex w)
{
      FP_CONTRACT_OFF
      double a, b, c, d, ac, bd, ad, bc, x, y;
      a = creal(z); b = cimag(z);
      c = creal(w); d = cimag(w);
      ac = a * c;  bd = b * d;
      ad = a * d;  bc = b * c;
      x = ac - bd;
      y = ad + bc;
```

[6] These properties are already implied for those cases covered in the tables, but are required for all cases (at least where the state for **`CX_LIMITED_RANGE`** is *off*).

```
      /* Recover infinities that computed as NaN+iNaN ... */
      if (isnan(x) && isnan(y)) {
         int recalc = 0;
         if (isinf(a) || isinf(b)) {  /* z is infinite */
            /* "Box" the infinity ... */
            a = copysign(isinf(a) ? 1.0 : 0.0, a);
            b = copysign(isinf(b) ? 1.0 : 0.0, b);
            /* Change NaNs in the other factor to 0 ... */
            if (isnan(c)) c = copysign(0.0, c);
            if (isnan(d)) d = copysign(0.0, d);
            recalc = 1;
         }
         if (isinf(c) || isinf(d)) {  /* w is infinite */
            /* "Box" the infinity ... */
            c = copysign(isinf(c) ? 1.0 : 0.0, c);
            d = copysign(isinf(d) ? 1.0 : 0.0, d);
            /* Change NaNs in the other factor to 0 ... */
            if (isnan(a)) a = copysign(0.0, a);
            if (isnan(b)) b = copysign(0.0, b);
            recalc = 1;
         }
         if (!recalc) {
            /* Recover infinities from overflow cases ... */
            if (isinf(ac) || isinf(bd) || isinf(ad) || isinf(bc)) {
               /* Change all NaNs to 0 ... */
               if (isnan(a)) a = copysign(0.0, a);
               if (isnan(b)) b = copysign(0.0, b);
               if (isnan(c)) c = copysign(0.0, c);
               if (isnan(d)) d = copysign(0.0, d);
               recalc = 1;
            }
         }
         if (recalc) {
            x = INFINITY * (a * c - b * d);
            y = INFINITY * (a * d + b * c);
         }
      }
      return x + I * y;
}
```

In ordinary (finite) cases, the cost to satisfy the infinity property for the `*` operator is only one `isnan` test. This implementation opts for performance over guarding against undue overflow and underflow.

2. Division of two **double complex** operands could be implemented as follows.

```
#include <math.h>
#include <complex.h>
/* isinf is as in example 1 above */

/* Divide z / w ... */
double complex _Cdivd(double complex z, double complex w)
{
      FP_CONTRACT_OFF
      double a, b, c, d, logbw, denom, x, y;
      long llogbw = 0;
      a = creal(z); b = cimag(z);
      c = creal(w); d = cimag(w);
      logbw = logb(fmax(fabs(c), fabs(d)));
      if (isfinite(logbw)) {
         llogbw = (long)logbw;
         c = scalb(c, -llogbw);
         d = scalb(d, -llogbw);
      }
      denom = c * c + d * d;
      x = scalb((a * c + b * d) / denom, -llogbw);
      y = scalb((b * c - a * d) / denom, -llogbw);
      /*
       * Recover infinities and zeros that computed as NaN+iNaN;
       * the only cases are infinite/finite, finite/infinite,
       * and non-zero/zero ...
       */
      if (isnan(x) && isnan(y)) {
         if ((denom == 0.0) && (!isnan(a) || !isnan(b))) {
            x = copysign(INFINITY, c) * a;
            y = copysign(INFINITY, c) * b;
         }
         else if ((isinf(a) || isinf(b)) && isfinite(c) && isfinite(d)) {
            a = copysign(isinf(a) ? 1.0 : 0.0, a);
            b = copysign(isinf(b) ? 1.0 : 0.0, b);
            x = INFINITY * (a * c + b * d);
            y = INFINITY * (b * c - a * d);
         }
         else if (isinf(logbw) && isfinite(a) && isfinite(b)) {
            c = copysign(isinf(c) ? 1.0 : 0.0, c);
            d = copysign(isinf(d) ? 1.0 : 0.0, d);
            x = 0.0 * (a * c + b * d);
            y = 0.0 * (b * c - a * d);
         }
      }
      return x + I * y;
}
```

Scaling the denominator alleviates the main overflow and underflow problem, which is more serious than for multiplication. In the spirit of the multiplication example above, this code does not defend against overflow and underflow in the calculation of the numerator. Scaling with the **scalb** function, instead of with division, provides better roundoff characteristics.

Y.5.2 Additive operators

Semantics

If one operand has real type and the other operand has imaginary type, then the result has complex type. If both operands have imaginary type, then the result has imaginary type. (If either operand has complex type, then the result has complex type.)

In all cases the result and exception behavior of a **`+`** or **`-`** operator is defined by the usual mathematical formula:

±	*x*	*y*I*	*x + y*I*
u	*x ± u*	*±u + y*I*	*(x±u) + y*I*
*v*I*	*x ± v*I*	*(y±v)*I*	*x + (y±v)*I*
*u + v*I*	*(x±u) ± v*I*	*±u + (y±v)*I*	*(x±u) + (y±v)*I*

Y.6 `<complex.h>`

The macro

```
_Imaginary_I
```

has an imaginary type.

This subclause contains specification for the **`<complex.h>`** functions that is particularly suited to IEC 559 implementations.

The functions are continuous onto both sides of their branch cuts, taking into account the sign of zero. For example, **`csqrt(-2`** ± **`0*I) ==`** ± **`sqrt(2)*I`**

Since complex and imaginary values are composed of real values, each function may be regarded as computing real values from real values. Except as noted, the functions treat real infinities, NaNs, signed zeros, subnormals, and the exception flags in a manner consistent with the specification for real functions in X.9.

The functions **`conj`**, **`cimag`**, **`cproj`**, and **`creal`** are fully specified for all implementations, including IEC 559 ones, in 7.x.2. These functions raise no exceptions.

Each of the functions **`cabs`** and **`carg`** is specified by a formula in terms of a real function (whose special cases are covered in annex X):

cabs(x + i*y)	=	hypot(x, y)
carg(x + i*y)	=	atan2(y, x)

Each of the functions **`casin`**, **`catan`**, **`ccos`**, **`csin`**, **`ctan`**, and **`cpow`** is specified implicitly by a formula in terms of other complex functions (whose special cases are specified below):

casin(z)	=	-i*casinh(i*z)
catan(z)	=	-i*catanh(i*z)

ccos(z)	=	ccosh(i*z)
csin(z)	=	-i*csinh(i*z)
ctan(z)	=	-i*ctanh(i*z)
cpow(z, c)	=	cexp(c * clog(z))

For the other functions, the following subclauses specify behavior for special cases, including treatment of the invalid and divide-by-zero exceptions. For a function f satisfying f(conj(z)) = conj(f(z)), the specification for the upper half-plane implies the specification for the lower half-plane; if also the function f is either even, f(-z) = f(z), or odd, f(-z) = -f(z), then the specification for the first quadrant implies the specification for the other three quadrants.

Y.6.1 The `cacos` function

- **`cacos`**(conj(z)) = conj(**`cacos`**(z)).
- **`cacos`**(±0+i0) returns π/2-i0.
- **`cacos`**(-∞+i∞) returns 3π/4-i∞.
- **`cacos`**(+∞+i∞) returns π/4-i∞.
- **`cacos`**(x+i∞) returns π/2-i∞, for finite x.
- **`cacos`**(-∞+iy) returns π-i∞, for positive-signed finite y.
- **`cacos`**(+∞+iy) returns +0-i∞, for positive-signed finite y.
- **`cacos`**(±∞+iNaN) returns NaN±i∞ (where the sign of the imaginary part of the result is unspecified).
- **`cacos`**(±0+iNaN) returns π/2+iNaN.
- **`cacos`**(NaN+i∞) returns NaN-i∞.
- **`cacos`**(x+iNaN) returns NaN+iNaN and optionally raises the invalid exception, for nonzero finite x.
- **`cacos`**(NaN+iy) returns NaN+iNaN and optionally raises the invalid exception, for finite y.
- **`cacos`**(NaN+iNaN) returns NaN+iNaN.

Y.6.2 The `cacosh` function

- **`cacosh`**(conj(z)) = conj(**`cacosh`**(z)).
- **`cacosh`**(±0+i0) returns +0+iπ/2.
- **`cacosh`**(-∞+i∞) returns +∞+i3π/4.
- **`cacosh`**(+∞+i∞) returns +∞+iπ/4.
- **`cacosh`**(x+i∞) returns +∞+iπ/2, for finite x.
- **`cacosh`**(-∞+iy) returns +∞+iπ, for positive-signed finite y.
- **`cacosh`**(+∞+iy) returns +∞+i0, for positive-signed finite y.
- **`cacosh`**(NaN+i∞) returns +∞+iNaN.
- **`cacosh`**(±∞+iNaN) returns +∞+iNaN.
- **`cacosh`**(x+iNaN) returns NaN+iNaN and optionally raises the invalid exception, for finite x.
- **`cacosh`**(NaN+iy) returns NaN+iNaN and optionally raises the invalid exception, for finite y.
- **`cacosh`**(NaN+iNaN) returns NaN+iNaN.

Y.6.3 The `casinh` function

- **`casinh`**(conj(z)) = conj(**`casinh`**(z)) and **`casinh`** is odd.
- **`casinh`**(+0+i0) returns 0+i0.
- **`casinh`**(∞+i∞) returns +∞+iπ/4.

- **casinh**(x+i∞) returns +∞+iπ/2 for positive-signed finite x.
- **casinh**(+∞+iy) returns +∞+i0 for positive-signed finite y.
- **casinh**(NaN+i∞) returns ±∞+iNaN (where the sign of the real part of the result is unspecified).
- **casinh**(+∞+iNaN) returns +∞+iNaN.
- **casinh**(NaN+i0) returns NaN+i0.
- **casinh**(NaN+iy) returns NaN+iNaN and optionally raises the invalid exception, for finite nonzero y.
- **casinh**(x+iNaN) returns NaN+iNaN and optionally raises the invalid exception, for finite x.
- **casinh**(NaN+iNaN) returns NaN+iNaN.

Y.6.4 The `catanh` function

- **catanh**(conj(z)) = conj(**catanh**(z)) and **catanh** is odd.
- **catanh**(+0+i0) returns +0+i0.
- **catanh**(+∞+i∞) returns +0+iπ/2.
- **catanh**(+∞+iy) returns +0+iπ/2, for finite positive-signed y.
- **catanh**(x+i∞) returns +0+iπ/2, for finite positive-signed x.
- **catanh**(+0+iNaN) returns +0+iNaN.
- **catanh**(NaN+i∞) returns ±0+iπ/2 (where the sign of the real part of the result is unspecified).
- **catanh**(+∞+iNaN) returns +0+iNaN.
- **catanh**(NaN+iy) returns NaN+iNaN and optionally raises the invalid exception, for finite y.
- **catanh**(x+iNaN) returns NaN+iNaN and optionally raises the invalid exception, for nonzero finite x.
- **catanh**(NaN+iNaN) returns NaN+iNaN.

Y. 6.5 The `ccosh` function

- **ccosh**(conj(z)) = conj(**ccosh**(z)) and **ccosh** is even.
- **ccosh**(+0+i0) returns 1+i0.
- **ccosh**(+0+i∞) returns NaN±i0 (where the sign of the imaginary part of the result is unspecified) and raises the invalid exception.
- **ccosh**(+∞+i0) returns +∞+i0.
- **ccosh**(+∞+i∞) returns +∞+iNaN and raises the invalid exception.
- **ccosh**(x+i∞) returns NaN+iNaN and raises the invalid exception, for finite nonzero x.
- **ccosh**(+∞+iy) returns (+∞)*cis(y), for finite nonzero y. [7]
- **ccosh**(+0+iNaN) returns NaN±i0 (where the sign of the imaginary part of the result is unspecified).
- **ccosh**(+∞+iNaN) returns +∞+iNaN.
- **ccosh**(x+iNaN) returns NaN+iNaN and optionally raises the invalid exception, for finite nonzero x.
- **ccosh**(NaN+i0) returns NaN±i0 (where the sign of the imaginary part of the result is unspecified).
- **ccosh**(NaN+iy) returns NaN+iNaN and optionally raises the invalid exception, for all nonzero numbers y.
- **ccosh**(NaN+iNaN) returns NaN+iNaN.

[7] cis(y) is defined by cos(y) + i*sin(y).

Y.6.6 The `csinh` function

- **`csinh`**(conj(z)) = conj(**`csinh`**(z)) and **`csinh`** is odd.
- **`csinh`**(+0+i0) returns +0+i0.
- **`csinh`**(+0+i∞) returns ±0+iNaN (where the sign of the real part of the result is unspecified) and raises the invalid exception.
- **`csinh`**(+∞+i0) returns +∞+i0.
- **`csinh`**(+∞+i∞) returns ±∞+iNaN (where the sign of the real part of the result is unspecified) and raises the invalid exception.
- **`csinh`**(+∞+iy) returns (+∞)*cis(y), for positive finite y.
- **`csinh`**(x+i∞) returns NaN+iNaN and raises the invalid exception, for positive finite x.
- **`csinh`**(+0+iNaN) returns ±0+iNaN (where the sign of the real part of the result is unspecified).
- **`csinh`**(+∞+iNaN) returns ±∞+iNaN (where the sign of the real part of the result is unspecified).
- **`csinh`**(x+iNaN) returns NaN+iNaN and optionally raises the invalid exception, for finite nonzero x.
- **`csinh`**(NaN+i0) returns NaN+i0.
- **`csinh`**(NaN+iy) returns NaN+iNaN and optionally raises the invalid exception, for all nonzero numbers y.
- **`csinh`**(NaN+iNaN) returns NaN+iNaN.

Y.6.7 The `ctanh` function

- **`ctanh`**(conj(z)) = conj(**`ctanh`**(z)) and **`ctanh`** is odd.
- **`ctanh`**(+0+i0) returns +0+i0.
- **`ctanh`**(+∞+iy) returns 1+i0, for all positive-signed numbers y.
- **`ctanh`**(x+i∞) returns NaN+iNaN and raises the invalid exception, for finite x.
- **`ctanh`**(+∞+iNaN) returns 1±i0 (where the sign of the imaginary part of the result is unspecified).
- **`ctanh`**(NaN+i0) returns NaN+i0.
- **`ctanh`**(NaN+iy) returns NaN+iNaN and optionally raises the invalid exception, for all nonzero numbers y.
- **`ctanh`**(x+iNaN) returns NaN+iNaN and optionally raises the invalid exception, for finite x.
- **`ctanh`**(NaN+iNaN) returns NaN+iNaN.

Y.6.8 The `cexp` function

- **`cexp`**(conj(z)) = conj(**`cexp`**(z)).
- **`cexp`**(±0+i0) returns 1+i0.
- **`cexp`**(+∞+i0) returns +∞+i0.
- **`cexp`**(-∞+i∞) returns ±0±i0 (where the signs of the real and imaginary parts of the result are unspecified).
- **`cexp`**(+∞+i∞) returns ±∞+iNaN and raises the invalid exception (where the signs of the real and imaginary parts of the result are unspecified).
- **`cexp`**(x+i∞) returns NaN+iNaN and raises the invalid exception, for finite x.
- **`cexp`**(-∞+iy) returns +0*cis(y), for finite y.
- **`cexp`**(+∞+iy) returns +∞*cis(y), for finite nonzero y.
- **`cexp`** (-∞+iNaN) returns ±0±i0 (where the signs of the real and imaginary parts of the result are unspecified).
- **`cexp`** (+∞+iNaN) returns ±∞+iNaN (where the sign of the real part of the result is unspecified).

- **`cexp`** (NaN+i0) returns NaN+i0.
- **`cexp`** (NaN+iy) returns NaN+iNaN and optionally raises the invalid exception, for all nonzero numbers y.
- **`cexp`** (x+iNaN) returns NaN+iNaN and optionally raises the invalid exception, for finite x.
- **`cexp`** (NaN+iNaN) returns NaN+iNaN.

Y.6.9 The `clog` function

- **`clog`**(conj(z)) = conj(**`clog`**(z)).
- **`clog`**(-0+i0) returns -∞+iπ and raises the divide-by-zero exception.
- **`clog`**(+0+i0) returns -∞+i0 and raises the divide-by-zero exception.
- **`clog`**(-∞+i∞) returns +∞+i3π/4.
- **`clog`**(+∞+i∞) returns +∞+iπ/4.
- **`clog`**(x+i∞) returns +∞+iπ/2, for finite x.
- **`clog`**(-∞+iy) returns +∞+iπ, for finite positive-signed y.
- **`clog`**(+∞+iy) returns +∞+i0, for finite positive-signed y.
- **`clog`**(±∞+iNaN) returns +∞+iNaN.
- **`clog`**(NaN+i∞) returns +∞+iNaN.
- **`clog`**(x+iNaN) returns NaN+iNaN and optionally raises the invalid exception, for finite x.
- **`clog`**(NaN+iy) returns NaN+iNaN and optionally raises the invalid exception, for finite y.
- **`clog`** (NaN+iNaN) returns NaN+iNaN.

Y.6.10 The `csqrt` function

- **`csqrt`**(conj(z)) = conj(**`csqrt`**(z)).
- **`csqrt`**(±0+i0) returns +0+i0.
- **`csqrt`**(-∞+iy) returns +0+i∞, for finite positive-signed y.
- **`csqrt`**(+∞+iy) returns +∞+i0, for finite positive-signed y.
- **`csqrt`**(x+i∞) returns +∞+i∞, for all x (including NaN).
- **`csqrt`**(-∞+iNaN) returns NaN±i∞ (where the sign of the imaginary part of the result is unspecified).
- **`csqrt`**(+∞+iNaN) returns +∞+iNaN.
- **`csqrt`**(x+iNaN) returns NaN+iNaN and optionally raises the invalid exception, for finite x.
- **`csqrt`**(NaN+iy) returns NaN+iNaN and optionally raises the invalid exception, for finite y.
- **`csqrt`**(NaN+iNaN) returns NaN+iNaN.