# C9X Complex Arithmetic Proposal

**WG14/N557 X3J11/96-021 (Draft 5/11/96)**

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

# Overview

This is a detailed proposal for integrating complex arithmetic into C9X. The specification is based on the "Complex C Extensions" chapter of X3J11's *Numerical C Technical Report* (X3/TR-17), with subsequent modifications in the directions approved by SC22/WG14. The main sections, in addition to this overview are:

C9X Edits
Complex Arithmetic <complex.h>
Complex Arithmetic for IEC 559 Implementations

### Background

At its initial meeting in 1989, the Numerical C Extensions Group (NCEG), now incorporated into the ANSI C language committee (X3J11), targeted complex arithmetic as one of the areas requiring supporting extensions, in order for C to become suitable for general numerical programming.

The first NCEG complex proposal, "Complex Extension to C" (Knaak), followed a traditional FORTRAN-like approach, adding complex data types as ordered pairs of real numbers and specifying the arithmetic of these types.

"Augmenting a Programming Language with Complex Arithmetic" (Kahan and Thomas) showed certain semantic and efficiency problems inherent in the traditional approach, including fundamental incompatibilities with advances in IEC 559 standard real arithmetic. To solve these problems, a second proposal, "Complex C Extensions" (Thomas), refined the traditional approach by adding both complex and imaginary types and specifying arithmetic on any combinations of real, imaginary, and complex operands. "Issues Regarding Imaginary Types for C and C++" (Thomas and Coonen) provided detailed rationale. At its June 1994 meeting, X3J11 approved following the direction of this proposal. At its December 1994 meeting, X3J11 accepted the specification for inclusion in its Technical Report. Subsequently, this document was recast for integration into C9X.

Through 1995, however, both the "Complex C Extensions" specification from the Technical Report and the traditional FORTRAN-like specification, now titled "Complex Type Simplified" (MacDonald and Knaak), were under consideration for C9X. At its February 1996 meeting, X3J11 expressed a preference for a base specification without imaginary types, and a specification with imaginary types for IEC 559 implementations. The current document, derived from the "Complex C Extensions" Technical Report, meets this desire by factoring the imaginary types into an annex for IEC 559. It was developed in close consultation from the authors of "Complex Type Simplified".

Following the lead of "Floating-Point Proposal for C9X" (N546), in the post-Irvine mailing, this proposal does not depend on overloading of math library functions. (At the February 1996 Irvine meetng, X3J11 expressed a desire for an updated floating-point proposal with overloading removed for separate consideration.)

**Substantive changes from Technical Report**

Specification for IEC 559 implementations has been added, including treatment of infinities. (The specification of special cases for library functions is still in progress.) *Rationale*: This was needed to support a programming model that takes advantage of IEC 559 features.

Support for the imaginary types has been moved to an annex for IEC 559 implementations. *Rationale*: This presents a simpler base standard for non-IEC 559 implementations.

The result type for a multiplicative opterator with two imaginary operands has been changed from real to complex. *Rationale*: This provides better compatibility between implementations that do and those that do not support imaginary types.

The macro scheme for the imaginary unit has been improved to not require a keyword in the implementation's name space. *Rationale*: This avoids defining a keyword.

The overloading specification has been removed. *Rationale*: X3J11 wants to consider overloading of math (and complex) library functions separately.

Imaginary and complex expressions have been disallowed as arguments for prefix increment and decrement operators. *Rationale*: Allowing an imaginary or complex operand for these operators seems potentially confusing and not particularly useful.

The imaginary and complex types are now designated without an underscore character, e.g. **`double complex`** instead of **`double_complex`**. *Rationale*: This is a better fit for the C standard. The underscore approach was intended to allow C++ compatible code, which now requires more extreme measures for other reasons.

A translation directive has been added to allow the implementation to use multiplication and division code based on the usual mathematical formulas. *Rationale*: This is to meet application needs for highest performance in cases where the range of values is known to be modest.

**Acknowledgments**

People who have made especially substantial contributions to this document include, in alphabetical order: Jerome Coonen, W. Kahan, David Knaak, Clayton Lewis, David Prosser, and Fred Tydeman.

Others who have provided invaluable contributions, reviews, or administrative help include, in alphabetical order: Frank Farance, Harry H. Cheng, Yinsun Feng, W.M. Gentleman, Bill Gibbons, Ron Guilmette, Doug Gwyn, David Hough, Rex Jaeschke,

David Keaton, John Kwan, Tom MacDonald, Stuart McDonald, Colin McMaster, Randy Meyer, Randy Meyers, Adolfo Nemirovsky, Jon Okada, P.J. Plauger, Thomas Plum, Pat Ricci, and Eric Yeh.

# C9X Edits

This section specifies editorial changes for the C9X draft in order to support complex arithmetic.

### 6.1.1 Keywords

*In Syntax, add the keyword:*

**`complex`**

*Append to Semantics this paragraph:*

The token **`complex`** becomes a keyword when the header **`<complex.h>`** is included, and not before.

### 6.1.2.5 Types

*In the first sentence of the sixth paragraph, replace "floating types" with "real floating types".*

*After the sixth paragraph, insert these paragraphs:*

There are three *complex types*, designated as **`float complex`**, **`double complex`**, and **`long double complex`**. Each is represented internally by a contiguous pair of representations of the *corresponding real type*, aligned in an implementation-defined manner identical to the real type. The low-address and high-address parts of the pair represent the real part and imaginary part, respectively, of the complex value.

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

> [The underlying implementation of the complex types is Cartesian, rather than polar, for overall efficiency and consistency with other programming languages. The implementation is explicitly stated so that characteristics and behaviors can be defined simply and unambiguously.
>
> An alternative would have been to introduce more types, **`float_complex`**, **`double_complex`**, etc., which would have better suited implementation as a C++ library. However, it would not have fit C so well, and the draft C++ standard uses template designations for complex classes so that typedefs would be required for porting between C and C++ anyway.]

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

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

*After the first sentence in the thirteenth paragraph (starting "The type char, ..."), insert:*

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

*In the last sentence in the thirteenth paragraph replace "*floating types*" with "*real floating types*".*

*After the fourteenth paragraph 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 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 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*, which is also the real type for the result, or the corresponding real type for a non-real result. The conversion of an operand yields a type in the same type-domain as the operand. The type-domain of the result type is as specified in the semantics for the operator; a real result has the common real type; each part of a non-real result has the common real type. This pattern is called the *usual arithmetic conversions*:

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

Otherwise, if the type, or corresponding real type, of either operand is `float`, the other operand is converted, without change of type-domain, to have the type, or corresponding real type, `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).

[Automatic conversion of non-complex operands to complex would require extra computation, while producing undesirable results in certain cases involving infinities, NaNs, and signed zeros. For example, with automatic conversion to complex,

$$
\begin{array}{lll}
2.0 * (3.0 + \infty i) & \Rightarrow & (2.0 + 0.0i) * (3.0 + \infty i) \\
 & \Rightarrow & (2.0*3.0 - 0.0*\infty) + (2.0*\infty + 0.0*3.0)i \\
 & \Rightarrow & \mathrm{NaN} + \infty i
\end{array}
$$

rather than the desired result, $6.0 + \infty i$.

Optimizers for implementations with infinities—including all IEC 559 ones—would not be able to eliminate the operations with the zero imaginary part of the converted operand.

The following example illustrates the problem pertaining to signed zeros; [6] explains why it matters. With automatic conversion to complex,

$$
\begin{array}{lll}
2.0 * (3.0 - 0.0i) & \Rightarrow & (2.0 + 0.0i) * (3.0 - 0.0i) \\
 & \Rightarrow & (2.0*3.0 + 0.0*0.0) + (-2.0*0.0 + 0.0*3.0)i \\
 & \Rightarrow & 6.0 + 0.0i
\end{array}
$$

rather than the desired result, $6.0 - 0.0i$.

The problems illustrated in the examples above have counterparts for pure imaginary operands. The mathematical product $2.0i * (\infty + 3.0i)$ should yield $-6.0 + \infty i$. With automatic conversion to complex,

$$
\begin{array}{lll}
2.0i * (\infty + 3.0i) & \Rightarrow & (0.0 + 2.0i) * (\infty + 3.0i) \\
 & \Rightarrow & (0.0*\infty - 2.0*3.0) + (0.0*3.0 + 2.0*\infty)i \\
 & \Rightarrow & \mathrm{NaN} + \infty i
\end{array}
$$

Annex X addresses such problems.

### 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 Constraints, replace* "arithmetic" *with* "real".

[Some mathematical practice would be supported by defining the relational operators for complex operands so that **z1** *op* **z2** would be true if and only if both **real(z1)** *op* **real(z2)** and also **imag(z1) == imag(z2)**. Believing such use to be uncommon, the committee voted against including this specification.]

### 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.

[For example,

```
0 == -0.0*I
```

is true, because (1) the usual arithmetic conversions promote the integer **0** to **double** (to match the other operand), (2) the values 0.0 and -0.0**I* convert to **double complex** values with all parts zero, and (3) -0.0 equals 0.0 arithmetically, even if not bitwise.]

### 5.2.4.2.2 Characteristics of floating types **<float.h>**

*In the second sentence of the paragraph defining* `FLT_EVAL_METHOD` *(from FPCE->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*".*

[Allowing a complex operand for increment and decrement operators seems potentially confusing and not particularly useful.]

### 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 "***`doubledouble`***".*

# 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 functions with the same name but with **`f`** and **`l`** suffixes which, respectively, are corresponding functions with **`float complex`** and **`long double complex`** parameters and **`float complex`** or **`float`** and **`long double complex`** or **`long double`** return values.

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. The macro is suitable for use in constant expressions.[2]

The macro

```
I
```

is defined to be **`_Imaginary_I`**.

> [Positing such a constant is a natural analog to the mathematical notion of augmenting the reals with the imaginary unit. It allows writing expressions in common mathematical style, for example **`x + y*I`**. Note that the multiplication here affects translated code, but does not necessitate an actual floating-point multiply, nor does the addition necessitate a floating-point add.
>
> The choice of **`I`** instead of **`i`** concedes to the widespread use of the identifier **`i`** for other purposes.
>
> The scheme for defining **`I`** and **`_Imaginary_I`** establishes a degree of uniformity in the designation of the imaginary unit as **`I`**, but offers flexibility to use a different identifier for the imaginary unit and to use **`I`** for other purposes. For example, one might follow the inclusion of **`<complex.h>`** with
>
> ```
> #define j _Imaginary_I
> #undef I
> ```
>
> An **`i`** suffix to designate imaginary constants is not required. Multiplication by **`I`** provides a sufficiently convenient and more generally useful notation for imaginary terms.]

---

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

## 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 mathematical formulas for multiplication and division are acceptable.[3] Each macro can occur either outside external declarations or preceding all 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*.

> [Calculated with the usual mathematical formula for multiplication, the result of raising any infinity to the fourth power has NaNs in both parts. The sum-of-squares denominator in the usual mathematical formula for division is particularly prone to overflow and underflow.
> Speed is critical in certain important, large matrix codes, which are known to require only modest range; even one test per multiplication may be too costly.]

## 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 X, 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 map a cut so the function is continuous as the cut is approached coming around the finite endpoint of the cut in a counter

[3] 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.

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.

**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 $[0, \pi]$ 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 $[-\pi/2, \pi/2]$ 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 [-$\pi$/2, $\pi$/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 $[-i\pi/2, i\pi/2]$ 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 $[-i\pi/2, i\pi/2]$ 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.

### 7.x.2.17 The cabs function

**Synopsis**

```
#include <complex.h>
double complex 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 $x^y$, with a branch cut for the first parameter along the negative real axis.

### Returns

The **cpow** function returns the complex power function $x^y$.

### 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 c**imag** function computes the imaginary part of **z**.

#### Returns

The **cimag** function returns the imaginary part of **z**.

### 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 infinities (even ones with one infinite part and one NaN part) project to positive infinity on the real axis.

#### Returns

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

[Two topologies are commonly used in complex mathematics: the complex plane with its continuum of infinities and the Riemann sphere with its single infinity. The complex plane is better suited for transcendental functions, the Riemann sphere for algebraic functions. The complex types with their multiplicity of infinities provide a useful (though imperfect) model for the complex plane. The **cproj** function helps model the Riemann sphere by mapping all infinities to one, and should be used just before any operation, especially comparisons, that might give spurious results for any of the other infinities.

Note that a complex value with one infinite part and one NaN part is regarded as an infinity, not a NaN, because if one part is infinite, the complex value is infinite independent of the value of the other part. For the same reason, **cabs** returns an infinity if its argument has an infinite part and a NaN part.]

### 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`.

# Complex Arithmetic for IEC 559 Implementations
**An extension to Annex X: IEC 559 Floating-Point Arithmetic**

This section extends Annex X proposed as part of the floating-point specification for C9X in order to specify complex arithmetic for IEC 559 implementations.

## X.10 Complex arithmetic

### X.10.1 Keywords

The syntax in subclause 6.1.1 is extended to include the keyword

**`imaginary`**

The token **`imaginary`** becomes a keywords when the header **`<complex.h>`** is included, and not before.

### X.10.2 Types

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

> [Although not present in older complex arithmetic facilities, e.g. FORTRAN's, the imaginary types naturally model the imaginary axis of complex analysis, promote computational and storage efficiency, and help capture the completeness and consistency of IEC 559 arithmetic for the complex domain. See [6], [12], [15], and rationale in 6.2.1.5 for more discussion of imaginary types.
>
> The example of the mathematical product 2.0i * (∞ + 3.0i), which should yield -6.0 + ∞i, demonstrates the need for imaginary types. Without them, 2.0*i* would have to be represented as 0.0 + 2.0*i*, implying that NaN + ∞*i* would be the semantically correct result—regardless of conversion rules. And optimizers for implementations with infinities—including all IEC 559 ones—would not be able to eliminate the operations with the zero real part.
>
> In general, the imaginary types, together with the conversion rules and operator specifications (below), allow substantially more efficient implementation. For example, multiplication of a real or imaginary by a complex can be implemented straightforwardly with two multiplications, instead of four multiplications and two additions.
>
> Some programs are expected to use the imaginary types implicitly in constructions with the imaginary unit **`I`**, such as **`x + y*I`**, and not explicitly in declarations. This suggests making the imaginary types private to the implementation and not available for explicit program declarations. However, such an approach was rejected as being less in the open spirit of C, and not much simpler. For the same reasons, the approach of treating imaginary-ness as an attribute of certain complex expressions, rather than as additional types, was rejected.
>
> The efficiency benefits of the imaginary types goes beyond what the implementation provides. In many cases programmers have foregone a programming language's complex arithmetic facilities, which, lacking an imaginary type, required contiguous storage of both

real and imaginary parts; programmers could store and manipulate complex values more efficiently using real arithmetic directly [15]. The imaginary types enable programmers to exploit the efficiency of the real formats, without having to give up support for complex arithmetic semantics.

Because of their representation and alignment requirements, imaginary arguments can be used like real arguments for **`fprintf`** and **`fscanf`**.]

## X.10.3 Conversions

### X.10.3.1 Imaginary types

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

### X.10.3.2 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 zero.

### X.10.3.3 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 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 corresponding real types.

## X.10.4 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 is required to satisfy certain properties (specified in X.10.4.1), but is not completely determined.

### X.10.4.1 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.)

[In previous versions of this specification the result of a multiplicative operator with two imaginary operands had real type. The change was motivated for compatibility between implementations with a complex-type imaginary unit `I` and those with a (pure) imaginary-type one (see X.10.5). Hence `I*I` will be complex for all implementations.]

If the operands are not both complex, then the result 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[4]:

- If one operand is an infinity and the other operand is a nonzero finite number, 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 number (and not a NaN) and the second operand is a zero, then the result of the `/` operator is an infinity.

[These properties, together with the `cproj` function in `<complex.h>` and the user paradigm of ignoring differences in infinities, largely supports the 1-infinity, Riemann sphere model for complex numbers.]

[4] 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*).

**Examples**

1. Multiplication of **`double complex`** operands could be implemented as follows. Note that the imaginary unit **`I`** has imaginary type (see X.10.5).

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

// To "box" infinities ...
static double complex box(double complex z)
{
      return copysign(isinf(real(z)) ? 1.0 : 0.0, real(z)) +
         I * copysign(isinf(imag(z)) ? 1.0 : 0.0, imag(z));
}

// Multiply z * w ...
double complex _Cmultd(double complex z, double complex w)
{
      double a, b, c, d, x, y;
      a = creal(z); b = cimag(z); c = creal(w); d = cimag(w);
      x = a * c - b * d;
      y = a * d + b * c;
      if (isnan(x) && isnan(y))
      {
         int recalc = 0;
         if (isinf(a) || isinf(b))  // if z is infinite
         {
            double complex zz;
            zz = box(z);
            a = creal(zz);
            b = cimag(zz);
            recalc = 1;
         }
         if (isinf(c) || isinf(d))  // if w is infinite
         {
            double complex ww;
            ww = box(w);
            c = creal(ww);
            d = cimag(ww);
            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.

> [Undue overflow for multiplication is less a problem than for division, because it occurs in more limited cases. Generally, undue underflow from complex multiplication is not a serious problem. Ideally, all multiplications and divisions would be correctly rounded, as

are the operations specified in the tables; however, the costs would be substantial, in the absence of special hardware support.]

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

```
#include <math.h>
#include <complex.h>
// box and isinf are as in example 1 above

// Divide z / w ...
double complex _Cdivd(double complex z, double complex w)
{
       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);
       if (isnan(x) && isnan(y))
       {
          double fact = 1;
          int recalc = 0;
          if (isinf(a) || isinf(b))  // if z is infinite
          {
             double complex zz;
             zz = box(z);
             fact = INFINITY;
             a = creal(zz);
             b = cimag(zz);
             recalc = 1;
          }

          if (logbw == INFINITY)  // if w is infinite
          {
             double complex ww;
             ww = box(w);
             fact /= INFINITY;
             c = creal(ww);
             d = cimag(ww);
             denom = c * c + d * d;
             recalc = 1;
          }
          if (recalc)
          {
             x = fact * scalb((a * c + b * d) / denom, -llogbw);
             y = fact * scalb((b * c - a * d) / denom, -llogbw);
          }
       }
       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.

> [Because of the roundoff characteristics preserved by the use of **`scalb`**, division of Gaussian integers is exact when it should be.]

### X.10.4.2 Additive operators

**Semantics**

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

| $\pm$ | $x$ | $y*I$ | $x + y*I$ |
|---|---|---|---|
| $u$ | $x \pm u$ | $\pm u + y*I$ | $(x \pm u) + y*I$ |
| $v*I$ | $x \pm v*I$ | $(y \pm v)*I$ | $x + (y \pm v)*I$ |
| $u + v*I$ | $(x \pm u) \pm v*I$ | $\pm u + (y \pm v)*I$ | $(x \pm u) + (y \pm v)*I$ |

> [The usual mathematical formulas for the **+** and **-** operators are inconsistent with the 1-infinity, Riemann sphere interpretation for complex numbers in that the sum of infinities can be an infinity, whereas a value with two NaN parts would be better. This is thought to be a relatively minor problem, because programs that add the results of two calculations that overflow generally have more serious problems.]

## X.10.5 <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, **`sqrt(-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. The functions treat real infinities, NaNs, signed zeros, and subnormals 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.

For other functions, the following subclauses specify behavior for special cases, including treatment of the invalid and divide-by-zero exceptions.[5]

| |
|---|
| Subclauses are required for cacos, casin, catan, ccos, csin, ctan, cacosh, casinh, catanh, ccosh, csinh, ctanh, cexp, clog, csqrt, cpow, cabs carg, cexp. Samples for casin and ccosh follow. |

### X.10.5.? The `casin` function

- **`casin`** is symmetric about the real and imaginary axes.
- **`casin(`**+∞+iNaN**`)`** returns NaN±i∞ (where the sign of the imaginary part is unspecified) and optionally raises the invalid exception.
- **`casin(`**+0+iNaN**`)`** returns +0+iNaN.
- **`casin(`**x+iNaN**`)`** returns NaN+iNaN, and optionally raises the invalid exception, for nonzero finite x.
- **`casin(`**NaN+i∞**`)`** returns NaN+i∞.
- **`casin(`**+0+i∞**`)`** returns +0+i∞.
- **`casin(`**x+i∞**`)`** returns NaN+i∞ and raises the invalid exception, for nonzero, finite or infinite x.
- **`casin(`**NaN+iy**`)`** returns NaN+iNaN and optionally raises the invalid exception, for finite y.
- **`casin(`**∞+i0**`)`** returns π/2+i∞.
- **`casin(`**∞+iy**`)`** returns NaN+i∞ and raises the invalid exception, for nonzero finite y.

### X.10.5.? The `ccosh` function

The following defines a portable implementation of the **`ccosh`** function.

```
double complex ccosh(double complex z)
{
      double x, y;
      x = creal(z);
      y = cimag(z);
      return cosh(x) * cos(y) + I * sinh(x) * sin(y);
}
```

[5] Sample implementations implicitly specify behavior for special cases.