WG14/N365 X3J11/94-050

Complex C Extensions

WG14/~~351~~ N365, X3J11/94-0~~36~~ 050

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

Author's Note

Until X3J11's June 1994 meeting, the effort to specify C extensions for complex arithmetic has focused for some time on the pivotal issue of imaginary types. Now that X3J11 has decided to proceed with imaginary types, we need to move on to a thorough review of all aspects of the specification. And time is short, with just two mailings between June and December when X3J11 is scheduled to conclude its technical report on numeric extensions. The purpose of this version is to collect as much review as possible, so that for the second mailing we can present a solid document which, after any minor changes indicated from the December meeting, can be included in the technical report.

To do:

- *provide more definition for <fp.h> overloads;*
- *evaluate additional functions for inclusion in library (norm, polar, cot's);*
- *resolve what arg(0) should be;*
- *add a foreword.*

Contributors and reviewers to this specification include Jerome Coonen, W.M. Gentleman, Bill Gibbons, David Hough, W. Kahan, David Keaton, David Knaak, Clayton Lewis, Tom MacDonald, Stuart McDonald, Adolfo Nemirovsky, Tom Plum, David Prosser, and Fred Tydeman.

1. Introduction

1.1 Purpose

This document specifies a set of extensions to the C programming language, designed to support complex arithmetic. The extensions are suitable for any implementation of Standard C [1], [2], augmented according to "Floating-Point C Extensions" [3], which the ANSI C committee X3J11 has approved for its technical report on numerical C extensions. For implementations supporting the IEEE floating-point standards [4], [5], these extensions allow complex arithmetic that is consistent with IEEE arithmetic.

This document is intended for the X3J11 technical report on numerical C extensions.

1.2 References

1. *International Standard Programming Languages—C*, (ISO/IEC 9899:1990 (E)).

2. *American National Standard for Information Systems*—Programming Language C (X3.159-1989).

3. "Floating-Point C Extensions", Jim Thomas (WG14/N350, X3J11/94-035—July 7, 1994).

4. *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985).

5. *IEEE Standard for Radix-Independent Floating-Point Arithmetic* (ANSI/IEEE Std 854-1987).

6. "Augmenting a Programming Language with Complex Arithmetic", W. Kahan and J. W. Thomas (NCEG/91-039—November 15, 1991).

7. "Complex Extension to C" (Revision 11), David Knaak (WG14/N335/X3J11./94-020—April 28, 1994).

8. "Imaginary Types for Complex Extensions to C", Jim Thomas (X3J11.1/92-071—October 24, 1992).

9. "Merging Complex and IEEE-754", Frederick John Tydeman (X3J11.1/92-061—November, 1992).

10. *Working Paper for Draft Proposed International Standard for Information Systems -- Programming Language C++* (X3J16/94-0098, WG21/N0485—May 26, 1994).

11. "What To Do About The Complex Extension To C?", David Knaak (X3J11.1/93-016—April 5, 1993).

12. "Issues Regarding Imaginary Types for C and C++", Jim Thomas and Jerome T. Coonen, *The Journal of C Language Translation*, Vol. 5, No. 3, Pages 134-138 (March 1994).

13. "A Proposal for Standard C++ Complex Number Classes", Al Vermeulen (X3J16/93-0165, WG21/N0372—1993).

For additional discussion about complex extensions, see NCEG documents 89-002, 89-026, 89-036, 89-038, 90-005, 90-006, 90-030, 91-005, 91-026, and 91-053; X3J11.1 documents 92-031, 93-020; and WG14/X3J11 document N362/94-047.

1.3 Organization of Document

This document follows the style conventions of [3]. The major subsections are:

1. this introduction;
2. language syntax, constraints, and semantics;
3. library facilities.

1.4 Definition of Terms

Other relevant terms are defined in [3].

- Complex type—a type for representing complex values. Any of the types `float_complex`, `double_complex`, or `long_double_complex`.
- Floating type—in this document, a real floating, imaginary, or complex type. In [1], [2], floating type refers to just real floating types.
- Imaginary type—a type for representing (pure) imaginary values. Any of the types `float_imaginary`, `double_imaginary`, or `long_double_imaginary`.
- Imaginary unit—the number *i* such that *i**i* = -1.
- Real floating type—one of the types `float`, `double`, or `long double`.
- Real type—an integral or real floating type.

2. LANGUAGE

2.1 Types (ISO §6.1.2.5; ANSI §3.1.2.5)

The integral types, together with the `float`, `double`, and `long double` types, are collectively called the *real types*.

Inclusion of the header `<complex.h>` enables designations for imaginary and complex types. The header defines macros

```
float_imaginary
double_imaginary
long_double_imaginary
float_complex
double_complex
long_double_complex
```

which expand to keywords in the implementation's name space (see §3.1).

There are three *imaginary types*, designated as:

```
float_imaginary
double_imaginary
long_double_imaginary
```

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

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 efficiency, and capture the completeness and consistency of IEEE arithmetic for the complex domain. See [6], [12], and rationale in §2.3.6 of this document for more discussion of imaginary types.

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

There are three *complex types*, designated as:

```
float_complex
double_complex
long_double_complex
```

Each is represented internally by a contiguous pair of representations of the corresponding real type. The first and second elements of the pair represent the real part and imaginary part, respectively, of the complex value.

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.

Standard C specification for basic and arithmetic types applies to the complex and imaginary types, except as specified otherwise.

An alternative would have been to introduce just two type macros, `complex` and `imaginary`, and to designate the complex and imaginary types as `float complex`, `float imaginary`, `double complex`, ... However, this approach would have precluded implementation as a C++ library and would have been a slightly less straightforward extension to C.

2.2 Constants

Inclusion of the header `<complex.h>` allows designation of an imaginary unit constant. The header defines a macro `I` which expands to `_Imaginary_I`, a keyword in the implementation's name space, denoting a constant with the imaginary type that corresponds to the minimum evaluation format [3] and with the value of the imaginary unit. For example, if the minimum evaluation format is `double`, then `I` has type `double_imaginary`.

Positing such a constant is a natural analog to the mathematical notion of augmenting the reals with the imaginary unit. It allows writing imaginary and complex expressions in common mathematical style, for example `x + y*I`. Note that the multiplication here affects translated code, but does not cause an actual floating-point multiply, nor does the addition cause a floating-point add.

The choice of `I` instead of `i` concedes to the widespread use of the identifier `i` for other purposes. The programmer can use a different identifier, say `j`, for the imaginary unit by following the inclusion of `<complex.h>` with

```
#undef I
#define j _Imaginary_I
```


274

The `i` suffix to designate imaginary constants, proposed in [7], is not required. Multiplication by `I` provides a sufficiently convenient and more generally useful notation for imaginary terms.

2.3 Conversions (ISO §6.2; ANSI §3.2)

This specification includes conversion between any complex type and any arithmetic type and between any imaginary type and any arithmetic type. Because of overloading considerations, the C++ complex class library can be expected to be somewhat more restrictive. For example, C++ may require an explicit function call to convert from a wider complex or imaginary type to a narrower one.

2.3.1 Imaginary types

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

2.3.2 Real and imaginary types

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

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

2.3.3 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 real types.

2.3.4 Real and complex types

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

2.3.5 Imaginary and complex types

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

2.3.6 Usual arithmetic conversions (ISO §6.2.1.5; ANSI §3.2.1.5)

The real parts of imaginary and complex operands and results follow the pattern of the usual arithmetic conversions for real types. Hence, for example, the product of a `float_complex` and a `double` or a `double_imaginary` entails a promotion of the complex operand to `double_complex`, and the semantic result type is `double_complex`.

Operations do not entail conversion of operands among real, imaginary, and complex types. For example, the product of a `double_imaginary` and a `float_complex` does not entail a conversion of the imaginary operand to complex.

Automatic conversion of real or imaginary operands to complex would require extra computation, while producing undesirable results in certain cases involving infinities and in certain cases involving signed zeros.

Examples

With automatic conversion to complex,

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

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

Note that if the semantics included automatic conversion to complex, then $NaN + \infty i$ would be the specified result. Hence optimizers for implementations with infinities—including all IEEE 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,

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

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

Imaginary types

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

$$\begin{aligned} 2.0i \cdot (\infty + 3.0i) &\Rightarrow (0.0 + 2.0i) * (\infty + 3.0i) \\ &\Rightarrow (0.0*\infty - 2.0*3.0) + (0.0*3.0 + 2.0*\infty)i \\ &\Rightarrow NaN + \infty i \end{aligned}$$

This also demonstrates the need for imaginary types. Without them, $2.0i$ would have to be represented as $0.0 + 2.0i$. implying $NaN + \infty i$ would be the semantically correct (though still undesirable) result—regardless of conversion rules. Optimizers for implementations with infinities—including all IEEE 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.

Most 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 as not simplifying much. For the same reasons, the approach of treating imaginary-ness as an attribute of certain complex expressions, rather than as additional types, was rejected.

Another approach, put forth in [11], would regard the special values—infinities, NaNs, and signed zeros—as outside the model. This would allow any behavior when special values occur, including much that is prescribed by this specification. However, this approach would not serve the growing majority of implementations, including all IEEE ones, supporting the special values. In order to provide a consistent extension of their treatment of special cases in real arithmetic, these implementations would require yet another specification in addition to the one suggested in [11]. On the other hand, implementations not supporting special values should have little additional trouble implementing imaginary types as proposed here.

2.4 Expressions (ISO §6.3; ANSI §3.3)

2.4.1 Multiplicative operators (ISO §6.3.5; ANSI §3.3.5)

If an expression that has real type is combined by multiplication or division with an expression that has imaginary type, then the result has imaginary type.

If two expressions that have imaginary type are multiplied or divided, then the result has real type.

If at least one operand expression of a multiplication or division has complex type, then the result has complex type.

The values of the imaginary and complex types are precisely the values of $y*I$ and $x + y*I$, respectively, where x and y are values of the corresponding real floating type and I is the value of the imaginary constant `I`. Hence, the following tables, taken from [6], describe the types and results of multiplications and divisions involving real, imaginary, and complex operands. x, y, u, v, s, and t denote real values.

Multiply

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

Divide

/	*x*	*y*I*	*x + y*I*
u	*x/u*	*(y/u)*I*	*(x/u) + (y/u)*I*
*v*I*	*(-x/v)*I*	*y/v*	*(y/v) + (-x/v)*I*
*u + v*I*	*s + t*I*	*s + t*I*	*s + t*I*

The computation of the product of two complex values should guard against undue overflow and underflow. [6] presents algorithms for the cells labeled *s + t*I*. Also see [9].

2.4.2 Additive operators (ISO §6.3.6; ANSI §3.3.6)

If an expression that has real type is combined by addition or subtraction with an expression that has imaginary type, then the result has complex type.

If two expressions that have imaginary type are added or subtracted, then the result has imaginary type.

If at least one operand expression of an addition or subtraction has complex type, then the result has complex type.

The following table, taken from [6], describes the types and results of addition and subtraction involving real, imaginary, and complex operands. *x*, *y*, *u*, and *v* denote real values.

Add/subtract

±	*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*

Note that some operations can be handled entirely (and correctly) at translation time, without floating-point arithmetic. Examples include *y * I*, *x + v * I*, and *I * I*.

2.4.3 Relational operators (ISO §6.3.8; ANSI §3.3.8)

The arithmetic operands of the relational operators are constrained to be 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)`. NCEG voted against including this specification.

2.4.4 Equality operators (ISO §6.3.9; ANSI §3.3.9)

Operands of the equality operators == and != can be expressions of any arithmetic type, including imaginary and complex types.

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 (including imaginary and

complex) are equal if and only if the results of their conversion to the complex type of width 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 the `double_complex` type as *0.0 + 0.0*I* and *0.0 - 0.0*I* , and (3) *-0.0* equals *0.0* arithmetically, even if not bitwise.

2.4.5 Expression evaluation methods

The expression evaluation methods described in [3] govern evaluation formats of expressions involving imaginary and complex types. For example, if the minimum evaluation format is `double`, then the product of two `float_complex` operands is represented in the `double_complex` format, and its parts are evaluated to `double`.

3. LIBRARIES

3.1 Complex Extensions <complex.h>

The header `<complex.h>` defines macros and functions that support complex arithmetic.

The macro

```
__COMPLEX__
```

expands to the constant `1`, indicating inclusion of `<complex.h>`.

The macros

```
float_imaginary
double_imaginary
long_double_imaginary
float_complex
double_complex
long_double_complex
```

expand to distinct keywords in the implementation's name space. For example,

```
#define float_complex _Float_complex
```

If the macro names later become standard keywords, then the implementor can remove the macro definitions from the header and equate the internal names to the macro names in the translator.

The macro

```
I
```

expands to `_Imaginary_I`, a keyword in the implementation's name space, denoting a constant with the imaginary type that corresponds to the minimum evaluation format and with the value of the imaginary unit.

See rationale in §2.2.

Lacking an imaginary type, [7] required macros in order to create certain special values. For example, an "imaginary" infinity could be created by `CMPLX(0.0,INFINITY)`. With the imaginary type, imaginary infinity is simply the value of `INFINITY*I`. ([3] defines the `INFINITY` macro.) And, in general, the values *y*I* and *x* + *y*I*, where *x* and *y* are real floating values, cover all values of the imaginary and complex types, hence eliminating this need for the complex macros.

The subsequent sections specify the declarations for functions involving complex and imaginary types. Semantic details, though outside the scope of this document, should be documented thoroughly by the implementation. [9] specifies semantics for IEEE implementations, and for other implementations with similar features.

3.1.1 Overloaded <fp.h> functions

When the header `<complex.h>` is included, some function designators from `<fp.h>` [3] become overloaded with additional prototypes having complex and imaginary parameters. The most common form for the extra prototypes is

```
floating-type function-designator ( complex-or-imaginary-type ) ;
```

Functions with extra prototypes of this form are:

`acos`	`cos`	`acosh`	`cosh`	`exp`
`asin`	`sin`	`asinh`	`sinh`	`log`
`atan`	`tan`	`atanh`	`tanh`	`sqrt`

For each of these functions, the implementation provides a set of prototypes—one for each complex and imaginary parameter type. For all these functions, if the parameter is complex then so is the return type. If the parameter is imaginary then the return type is real, imaginary, or complex, as appropriate for the particular function; in particular, if the parameter is imaginary, then the return types of `cos` and `cosh` are real, the return types of `sin`, `tan`, `sinh`, `tanh`, `asin`, and `atanh` are imaginary, and the return types of the others are complex. The format of the part(s) of the result is the wider of the format of the part(s) of the parameter type and the minimum evaluation format.

Example

If the minimum evaluation format were `double` then for `cos` the implementation would have

```
double_complex cos(float_complex);
double_complex cos(double_complex);
long_double_complex cos(long_double_complex);
double cos(float_imaginary);                    /* cos(x*I) == cosh(x) */
double cos(double_imaginary);
long double cos(long_double_imaginary);
```

Exploiting the fact that some functions map the imaginary axis onto the real or imaginary axis gains more efficient calculation involving imaginaries, and better meets user expectations in some cases. However, dropping out of the complex domain may lead to surprises as subsequent operations may be done with real functions, which generally are more restrictive than their complex counterparts. For example, `sqrt(-cos(I))` invokes the real `sqrt` function, which is invalid for the negative real value `-cos(I)`, whereas the complex `sqrt` is valid everywhere.

For a call to one of these functions with a complex or imaginary argument, the implementation chooses the prototype whose parameter matches the argument type.

This specification describes overload resolution rules in ad hoc fashion for various groups of functions. In fact, for the case without widest-need expression evaluation [3], these functions could be implemented for C++ with a set of overloaded prototypes in an ordinary library, where the complex and imaginary types are classes.

With widest-need expression evaluation [3], if the evaluation format passed down to the call is wider than the return-type parts of the chosen prototype, then the implementation chooses instead the prototype whose return-type parts match the wider format and whose parameter is of the same kind—complex or imaginary—as the argument.

Example

If the minimum evaluation format is `float`, then in

```
double d;
float_complex fc;
float_imaginary fi;
...
fc = d * sin(fi);
```

an implementation without widest-need would choose the `float_imaginary` prototype for `sin`. An implementation with widest-need would choose the `double_imaginary` prototype for `sin`.

The `fabs` function has extra prototypes of the form

`real-floating-type` `fabs(`*`complex-or-imaginary-type`*`)`

The implementation provides a set of prototypes—one for each complex and imaginary parameter type. The format of the real floating result is the wider of the formats of the part(s) of the argument and the minimum evaluation format [3]. The matching of calls to prototypes is analogous to the functions above.

The `pow` function has extra prototypes of these three forms:

```
complex-type pow(complex-or-imaginary-type, complex-or-imaginary-type)
complex-type pow(complex-or-imaginary-type, real-floating-type)
complex-type pow(real-floating-type, complex-or-imaginary-type)
```

Of the first form, there are 36 combinations of types for the two parameters, hence 36 prototypes.

Of the second form, there are 6 possible types for the first parameter. For each of these the implementation provides prototypes with the second parameter being the minimum evaluation format through all wider real floating-point formats.

Of the third form, there are 6 possible types for the second parameter. For each of these the implementation provides prototypes with the first parameter being the minimum evaluation format through all wider real floating point formats.

For all three forms, the format of the parts of the complex result matches the wider of the formats of the part(s) of the first and second parameters and the minimum evaluation format.

Example

If the minimum evaluation format is `long double`, then the implementation provides the 36 prototypes of the first form, all with return type `long_double_complex`. Of the second form it provides

```
long_double_complex pow(float_imaginary, long double);
long_double_complex pow(double_imaginary, long double);
long_double_complex pow(long_double_imaginary, long double);
long_double_complex pow(float_complex, long double);
long_double_complex pow(double_complex, long double);
long_double_complex pow(long_double_complex, long double);
```

and of the third form it provides

```
long_double_complex pow(long double, float_imaginary);
long_double_complex pow(long double, double_imaginary);
long_double_complex pow(long double, long_double_imaginary);
long_double_complex pow(long double, float_complex);
long_double_complex pow(long double, double_complex);
long_double_complex pow(long double, long_double_complex);
```

For a call to `pow` with at least one complex or imaginary argument, the implementation chooses a prototype as follows. First, any integer argument is converted to `double`. Then any real floating argument narrower than the minimum evaluation format is converted to the minimum evaluation format. Then the implementation chooses the prototype whose parameters match the argument types.

With widest-need expression evaluation [3], if the evaluation format passed down to the call is wider than the result-type parts of the chosen prototype, then the implementation chooses instead the prototype whose result-type parts match the wider format and whose parameters are of the same kind—real, imaginary, or complex—as the argument.

3.1.2 Complex-specific functions

The header `<complex.h>` declares functions pertaining specifically to complex arithmetic. They are overloaded to allow arguments of real, complex, or imaginary

type. Note that in the overloading forms below, *floating-type* includes complex and imaginary types (unlike in [3]).

3.1.2.1 The arg function

Synopsis

```
#include <complex.h>
real-floating-type arg(floating-type z);
```

Return types follow the same pattern as for `fabs` (§3.1.1).

Description

The `arg` function computes the argument or phase angle of `z`, in the range [-π, π]. Prototypes and matching of calls to prototypes follow the same pattern as for `fabs`.

Returns

The `arg` function returns the argument or phase angle of `z`, in the range [-π, π].

3.1.2.2 The conj function

Synopsis

```
#include <complex.h>
floating-type conj(floating-type z);
```

Return types match parameter types.

Description

The `conj` function computes the complex conjugate of `z`, by reversing the sign of its imaginary part, if any. A call is matched to the prototype whose type matches the argument type, after conversion of any integral argument to `double`.

Regarded as manipulation functions, `conj` and `imag`, `proj`, and `real` (below) have somewhat different prototype schemes than arithmetic functions like `sin`.

Returns

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

3.1.2.3 The imag function

Synopsis

```
#include <complex.h>
real-floating-type imag(floating-type z);
```

Return types correspond to the formats of the part(s) of the parameters. For example,

```
float imag(float_complex);
```

is one of the prototypes, regardless of the expression evaluation method.

Description

The `imag` function computes the imaginary part of `z`. A call is matched to the prototype whose parameter type matches the argument type, after conversion of any integral argument to the minimum evaluation format.

Returns

The `imag` function returns the imaginary part of `z`.

3.1.2.4 The proj function

Synopsis

```
#include <complex.h>
floating-type proj(floating-type z);
```

The return type is real if the parameter is real; the return type is complex if the parameter is complex or imaginary. The format of the part(s) of the return type match the format of the part(s) of the parameter.

Description

The `proj` 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. A call is matched to the prototype whose parameter type matches the argument type, after conversion of any integral argument to `double`.

Returns

The `proj` 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 `proj` 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 value is infinite independent of the value of the other part. For the same reason, `fabs` returns an infinity if its argument has an infinite part and a NaN part.

3.1.2.5 The real function

Synopsis

```
#include <complex.h>
real-floating-type real(floating-type z);
```

Return types follow the same pattern as for `imag` (§3.1.2.3).

Description

The `real` function computes the real part of `z`. Calls are matched to prototypes the same as for `imag`.

Returns

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