WG14/N335
X3J11/94-020

Complex Extension to C

(Revision 11)
WG14/N335
X3J11/94-020

David Knaak

Cray Research, Inc.
655F Lone Oak Drive
Eagan, MN 55121
knaak@cray.com or uunet!cray!knaak

Introduction

This document is a full specification for a proposed extension to Standard C that adds complex data types. The specification is given as modifications to the C Standard, document ANSI X3.159-1989, *American National Standard for Information Systems – Programming Language – C*, and the technically identical ISO/IEC 9899:1990, *Programming Languages – C*. This proposal is for consideration by X3J11 and possibly for consideration by a wider audience.

This document also includes rationale, summarizing some of the discussions related to this proposed extension. For additional discussion of various aspects of the extension and for discussion of some of the alternatives, see NCEG documents 89-002, 89-026, 89-036, 89-038, 89-039, 90-005, 90-006, 90-030, 91-005, 91-026, 91-039, 91-053, and X3J11.1 documents 92-031, 92-061, 92-071, 92-075, 93-020, and 93-049.

The extension described in this document overlaps somewhat with the extensions described in document WG14/N319,X3J11/94-003 ''Floating-Point C Extensions'' and in document X3J11.1/92-061 ''Complex and IEEE-754.'' This document, ''Complex Extension to C'', focuses on basic language issues for complex types and the basics of the complex elementary math functions. It does not specify behavior for the special values of signed zeros, NaNs, and infinities for those implementations that support them. ''Floating-Point C Extensions'' specifies extensions to facilitate a wide range of numerical programming but does not specifically address complex arithmetic. ''Complex and IEEE-754'' specifies extensions to the complex elementary math functions including the special values of signed zeros, NaNs, and infinities. Committee X3J11 has expressed a desire to merge these documents after any conflicts amongst the proposals are resolved.

This document does not include imaginary types as part of the proposed extension. The reasons for this are discussed in the rationale part of section **6.1.2.5** (**3.1.2.5**).

Sections that can be regarded as extending or modifying a particular section of the Standard C documents are marked with the section number from the ANSI C document. For example:

2.2.4.2 Numerical Limits

The corresponding section number in the ISO C document is three greater, that is, **5.2.4.2**.

259

5.2.4.2 (2.2.4.2) Numerical limits

Add on page 15 at line 34:

> The characteristics of each of the real and imaginary parts of **`float complex`**, **`double complex`**, and **`long double complex`** numbers are the same as for the corresponding real floating types.

RATIONALE:

The specification that the parts of the complex floating types have the same characteristics as the corresponding real floating types, including their numerical limits, implies that the underlying implementation of the complex floating types is Cartesian. With this requirement, the limits for the complex plane are clearly and easily defined in terms of the limits for real floating types. See rationale with section **6.1.2.5 (3.1.2.5)** for further discussion.

6.1.1 (3.1.1) Keywords

Add to the list of keywords on page 20, after line 19:

```
complex
```

Semantics

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

Add on page 20, after line 26:

Forward reference: complex mathematics (7.14) (4.14)

RATIONALE:

When the keyword **`complex`** becomes part of the standard, some existing codes that already use this name as an identifier will have to be modified. This keyword, when part of a common extension rather than as part of the standard, can be implemented in the following way to avoid the need for those using **`complex`** as an identifier (but not a type specifier) to have to modify code. The keyword shall be a name that is in the implementor's name space, such as **`_Complex`**. However, if the user wishes to use the word **`complex`** for this data type, the inclusion of the header **`<complex.h>`**, which contains a macro definition such as:

```
#define complex _Complex
```

is required. When **`complex`** becomes a standard keyword, then all that is needed is to remove this line from the header, and equate **`complex`** and **`_Complex`** in the translator.

6.1.2.5 (3.1.2.5) Types

Replace on page 24, line 9, the first sentence with:

> There are three *real floating types*, designated as **`float`**, **`double`**, and **`long double`**.

Add on page 24 after line 11:

> There are three *complex floating types*, designated as **`float complex`**, **`double complex`**, and **`long double complex`**. The real and imaginary parts of the complex floating types each have the same representation as the corresponding real

307

floating types. The set of values of the type **`float complex`** is a subset of the set of values of the type **`double complex`**, the set of values of the type **`double complex`** is a subset of the set of values of the type **`long double complex`**.

Add the following paragraph on page 24 after line 11:

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

RATIONALE:

The complex extension to C is intended to provide as direct a mapping as possible of complex mathematics (for example, complex algebra or complex analysis) onto the C language. This mapping should be as close as possible in both how it is expressed and the results that are obtained. Because of the inherent characteristics of the C language and of digital computer systems, this mapping may be less than perfect, but this mapping should still be the guiding principle for the extension.

The statement "The real and imaginary parts of the complex floating types each have the same representation as the corresponding real floating types." is the same as saying that the underlying implementation of the complex types is Cartesian. This implementation is explicitly stated so that behaviors can be defined simply and unambiguously. Without being explicit about the underlying implementation, describing limits, promotion rules, and the behavior of math functions would be very abstract. And so if one representation is to be selected, Cartesian is the most natural for virtually all modern computers. This specification is consistent with the Fortran 77 specification of complex types. Other topologies, such as polar, are useful for many problems but are not defined by this extension. When needed, users can define additional functions or macros in terms of the underlying cartesian implementation.

This extension does not include integral types. However, the addition of complex integral types at some future date would be upward compatible with the current proposal.

This extension does not include imaginary types. Addition of imaginary types was proposed to and discussed by the committee. The justification for imaginary types was that these are necessary in order to avoid generation of undesired NaNs and to avoid getting the sign of zero wrong when infinities and minus zeros are part of a complex operand. If imaginary types are added for these reasons then must not the language specification properly handle all the cases involving signed zeros, NaNs, and infinities? In other words, the justification could be stated more broadly as:

> The imaginary types are needed so that all the rules of IEEE floating point arithmetic can be completely, correctly, and consistently followed. That by following these rules, the sign of zero will always be correct, NaNs and infinities will be propagated correctly, no undeserved NaNs or infinities will be generated, and no undeserved underflows or overflows will occur. By following these rules, algorithms can be ported from one conforming implementation to another and yield identical results.

The statement on which the lack of imaginary types is based is:

> The imaginary types are not needed because in the vast majority of complex arithmetic cases the sign of zero does not matter and NaNs and infinities do not occur.

301

Wherever signed zeros do matter or where NaNs and infinities are expected to occur, they can be checked for. And if they occur unexpectedly, the less they propagate, the better.

The addition of an imaginary type would add a lot of rules to the language. For example:

- The type of the the product of 2 imaginary numbers is real.
- The type of the sum of 2 imaginary numbers is imaginary, unless the result is zero, in which case the type is real.

In summary, the costs of adding imaginary types is not justified by the benefits.

6.1.3 (3.1.3) Constants

6.1.3.1 (3.1.3.1) Floating Constants

Syntax

floating-constant:
real-floating-constant
complex-floating-constant

real-floating-constant:
fractional-constant exponent-part$_{opt}$ *real-suffix*$_{opt}$
digit-sequence exponent-part real-suffix$_{opt}$

complex-floating-constant:
fractional-constant exponent-part$_{opt}$ *complex-suffix*$_{opt}$
digit-sequence exponent-part complex-suffix$_{opt}$
digit-sequence imaginary-suffix

fractional-constant:
digit-sequence$_{opt}$ **.** *digit-sequence*
digit-sequence **.**

exponent-part:
e *sign*$_{opt}$ *digit-sequence*
E *sign*$_{opt}$ *digit-sequence*

sign: one of
+ **-**

digit-sequence:
digit
digit-sequence digit

302

complex-suffix:
float-suffix imaginary-suffix
imaginary-suffix float-suffix
long-suffix imaginary-suffix
imaginary-suffix long-suffix
imaginary-suffix

real-suffix:
float-suffix
long-suffix

float-suffix: one of
f F

long-suffix: one of
l L

imaginary-suffix:
i

Semantics

An unsuffixed floating constant has type **double**. If suffixed by the letter **f** or **F**, it has type **float**. If suffixed by the letter **l** or **L**, it has type **long double**. If suffixed by the letter **i** it has type **double complex**. If suffixed by both the letters **i** and **f** or **F**, it has type **float complex**. If suffixed by both the letters **i** and **l** or **L**, it has type **long double complex**. A floating constant that has a complex type specifies the value of the imaginary part, and the real part has the value zero. Any floating constant suffixed with **i** is called a *complex floating constant.* Any floating constant not suffixed with **i** is called a *real floating constant.*

RATIONALE:

Either **i** or **j** could be selected for the imaginary suffix used to form complex constants. And in fact, both could be allowed. However, allowing both would add unnecessary duplication to the language.

The production rule *digit-sequence imaginary-suffix* is purely for notational convenience and is not necessary for completeness. It allows the form **1i** to be used rather than the equivalent **1.i** or **1.0i**

Inclusion of the **<complex.h>** header is not necessary to define a complex constant because these constants cannot appear in any standard conforming program.

6.2 (3.2) CONVERSIONS

6.2.1 (3.2.1) Arithmetic operands

Add this sub-section to page 36 after line 15:

6.2.1.4.1 (3.2.1.4.1) Complex types

When a value of complex type is promoted or demoted to another complex type, both parts follow the same promotion and demotion rules as for the corresponding real types.

When a value of real type is promoted to a complex type, the real part of the complex value gets the same value as if the promotion was to the corresponding real type and the imaginary part of the complex value gets the value of positive zero (+0).

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

6.2.1.5 (3.2.1.5) Usual arithmetic conversions

Replace the section on page 36 starting at line 17 with:

All types have three type attributes called the ***dimension***, the ***format***, and the ***length***. The dimension attribute specifies whether the values of the type can be represented on a one dimensional line, (i.e., real numbers) or on a two dimensional plane, (i.e., complex numbers). The format attribute specifies whether the values of the type are represented with an exponent part, (i.e., floating numbers) or without an exponent part, (i.e., integral numbers). The length attribute specifies how many bits are used to represent the magnitude and precision of the type. The values for each of these attributes are ranked, from highest to lowest, as shown below. For example, complex ranks higher than real for the dimension attribute.

dimension	format	floating length	integral length

complex
real
	floating
	integral
		`long double`
		`double`
		`float`
			`unsigned long`
			`signed long`
			`unsigned int`
			`signed int`

Many binary operators that have operands of arithmetic types cause implicit conversions of one or both operands. The purpose of the conversions is to yield a common format and length for the two operands which is the type of the result. These implicit conversions of the operands are called the *usual arithmetic conversions*.

The conversions shall preserve the original magnitude and precision of both operands except that precision may be lost when an integral type is converted to a floating type. This will occur if the magnitude of the integer is too great for the mantissa of the floating type to represent it exactly.

The rules for the usual arithmetic conversions are:

1) The dimension of the result type is that of the higher ranking dimension of the operands.

2) The format of the result type is that of the higher ranking format of the operands.

3) If the format of the result type is floating then:
   the length of the result type is that of the higher ranking floating length of the operands.
   else the format of the result type is integral and:
   the integral promotions are performed on both operands, and the length of

the result type is that of the higher ranking integral length of the promoted operands with one exception. The exception is that if one operand has type **`signed long`** and the other has type **`unsigned int`** and if a **`signed long`** cannot represent all the values of an **`unsigned int`**, the length of the result is **`unsigned long`**.

4) After the result type is determined and before the binary operation is performed, any operand that does not already have the same type as the result type is promoted to the result type.

An implementation is not required to actually convert the operands as long as the result of the binary operation is the same as the result obtained if all the conversion rules were followed. Similarly, operands may be converted when not required as long as the result of the binary operation is the same as the result obtained if all the conversion rules were followed.

RATIONALE:

The conversion rules spell out an unambiguous path for determining the result type of a binary operator and for specifying any conversions to be done on the operands. However, if it is more efficient in a given implementation to take a different path to get the same result, that is allowed. The primary requirement is that the result must be the same ''as if'' all the arithmetic conversion rules were followed.

The standard specifies a set of promotion rules such that if the user writes code that assumes these promotions will be done, then the code will be portable across conforming implementations. An implementation may perform optimizations such that promotions are not done if the results for a conforming program are the same as if the promotions were done.

6.3 (3.3) EXPRESSIONS

6.3.2 (3.3.2) Postfix operators
6.3.3 (3.3.3) Unary operators
6.3.4 (3.3.4) Cast operators
6.3.5 (3.3.5) Multiplicative operators
6.3.6 (3.3.6) Additive operators
6.3.7 (3.3.7) Bitwise shift operators
6.3.10 (3.3.10) Bitwise AND operator
6.3.11 (3.3.11) Bitwise exclusive OR operator
6.3.12 (3.3.12) Bitwise inclusive OR operator
6.3.13 (3.3.13) Logical AND operator
6.3.14 (3.3.14) Logical OR operator
6.3.15 (3.3.15) Conditional operator

No change to the above sections.

RATIONALE:

A complex number is an arithmetic type and a scalar type. Therefore, all operators that apply to arithmetic and scalar types also apply to complex types. If complex integral types are ever added to the language, then the constraints for some, but not all, of the operators must change ''integral type'' to ''real integral type''.

The following are mathematical formulas for the basic complex operations:

305

$$z_1 + z_2 = (a + bi) + (c + di) = (a + c) + (b + d)i$$
$$z_1 - z_2 = (a + bi) - (c + di) = (a - c) + (b - d)i$$
$$z_1 * z_2 = (a + bi) * (c + di) = (ac - bd) + (bc + ad)i$$
$$z_1 \div z_2 = (a + bi) \div (c + di) = [(ac + bd) \div (c^2 + d^2)] + [(bc - bd) \div (c^2 + d^2)]i$$

The standard specifies what the result of an operation must be but does not specify how an implementation is to perform the operation. On digital computers with finite range of representable floating point values, operations such as complex division can lead to overflow or underflow of intermediate values even when the mathematical result is with the range of representable values. Unless the standard specifies an algorithm for all implementations to use, whether and when overflow or underflow occurs and the speed of the operation may vary from implementation to implementation.

If code is written for implementations that support special values such as signed zeros, NaNs, and infinities, and if the code is written with the assumption that these values could arise during execution of the program, then the user needs to know how that implementation performs these operations. And it is up to the user to decide the meaning of these special values in the context of the program.

6.3.8 (3.3.8) Relational operators

Change the first constraint on page 49, line 33 to:

both operands have real type;

Change the first sentence on page 49, line 37 to:

If both of the operands have real type, the usual arithmetic conversions are performed.

RATIONALE:

While equality and inequality have clear mathematical meaning for complex numbers, greater than or less than do not have any standard meaning. A user who wants to design an ordering for complex values, can write a macro or function, similar to the following:

```
#define C_LE(z1,z2) (cabs(z1) <= cabs(z2))
```

This gives the user control over the meaning of relational operations with complex operands.

6.3.9 (3.3.9) Equality operators

Add on page 50, after line 38:

For IEEE implementations, if a NaN appears in either part of either operand when comparing two complex numbers, they compare unequal.

6.4 (3.4) CONSTANT EXPRESSIONS

Change the sentence on page 56 starting on line 28 to:

An arithmetic constant expression shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants, and **`sizeof`** expressions whose operand does not have a *variable*

306

length array type (see WG14/N317,X3J11/94-001 ''Arrays of Variable Length'').

6.5 (3.5) DECLARATIONS

6.5.2 (3.5.2) Type specifiers

Add to the list of type-specifiers on page 59, after line 33:

```
complex
```

Add to the list on page 60, after line 17:

- **`float complex`**
- **`double complex`**
- **`long double complex`**

6.5.6 (3.5.6) Type definitions

Remove or modify the example on page 71, lines 19-30 because the example gives the name ''complex'' to a structure type.

7 (4) LIBRARY

Add the following sections to the end of page 177:

7.14 (4.14) Complex Mathematics <complex.h>

The header **`<complex.h>`** defines five macros, declares three typedefs, and declares several mathematical functions. These functions take **`double complex`** arguments and return **`double`** or **`double complex`** values as specified below. This header must be included to declare complex types if the complex types are just an extension to the language. Inclusion of the header is not necessary to create a complex constant. If the complex types are a formal part of the language, this header does not need to be included to declare complex types.

The functions are:

```
csin
ccos
cexp
clog
cpow
csqrt
cabs
cimag
conj
creal
```

The macros are:

```
complex
```

which expands into implementation-defined spelling for declaring complex types,

307

```
CMPLXF
CMPLX
CMPLXL
```

which are described in **7.14.1** (**4.14.1**), and

```
_STD_COMPLEX
```

which is equal to the decimal constant 1, intended to indicate that the header **<complex.h>** has been included.

The typedefs are:

```
typedef float complex float_complex;
typedef double complex double_complex;
typedef long double complex long_double_complex;
```

RATIONALE:

A programmer may want to write code that includes complex numbers and that is valid for both C and C++. Because the proposed standard C++ method to support complex numbers (WG21/N0372, X3J16/93-0165, WG14/N313, X3J11/93-060) defines complex classes rather than complex types, the syntax for C and C++ differs in some ways. By following a simple set of rules, a programmer can write code for complex numbers that is portable between C and C++.

1. Use the typedef names instead of the type names. These typedef names are the same as the proposed C++ class names.

2. Use the **CMPLX** macros instead of the **i** suffix for complex constants. The **CMPLX** macros will expand to C syntax if compiling for C and expand into C++ syntax if compiling for C++.

3. Use the **CMPLX** macros and the **creal** and the **cimag** functions to convert a complex value from a higher precision to a lower precision type.

4. If complex math functions are used, add macros such as

```
#if defined(_STD_COMPLEX) && defined(__cplusplus)
#define csin sin
#endif
```

7.14.1 (4.14.1) Complex Operators

7.14.1.1 (4.14.1.1) The CMPLXF Macro

Synopsis

```
#include <complex.h>
float complex CMPLXF(float x, float y);
```

Description

The **CMPLXF** macro behaves like an operator that creates a value with type **float complex**. If either argument is not of type **float**, it is first converted to type **float**.

Returns

The **CMPLXF** macro returns a value with type **float complex** whose real part is **x** and whose imaginary part is **y**.

7.14.1.2 (4.14.1.2) The CMPLX Macro

Synopsis

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

Description

The **CMPLX** macro behaves like an operator that creates a value with type **double complex**. If either argument is not of type **double**, it is first converted to type **double**.

Returns

The **CMPLX** macro returns a value with type **double complex** whose real part is **x** and whose imaginary part is **y**.

7.14.1.3 (4.14.1.3) The CMPLXL Macro

Synopsis

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

Description

The **CMPLXL** macro behaves like an operator that creates a value with type **long double complex**. If either argument is not of type **long double**, it is first converted to type **long double**.

Returns

The **CMPLXL** macro returns a value with type **long double complex** whose real part is **x** and whose imaginary part is **y**.

RATIONALE:

Various methods have been proposed for creating a complex number from two real numbers. One proposed method is to use imaginary constants as in the expression:

```
x + y * 1.0i
```

This method has the advantage of adding only a small amount of syntax to the language and is a compact form in many expressions. But IEEE implementations may require a way to create complex values where either the real or imaginary parts are special values such as +∞, -∞, and NaN, and this method does not always produce the desired complex result. For example, if `y` has the value +∞, then:

```
   y * 1.0i
=> (+∞) * (0.0, 1.0)
=> [(+∞ * 0.0), (+∞ * 1.0)]
=> (NaN, +∞)
```

when the desired result is:

```
(0.0, +∞)
```

Another proposed method is to use something similar to compound literals as the expression:

```
(double complex){ ._real = x, ._imag = y }
```

The full specification of compound literals is defined in document X3J11.1/93-039, "Designated Initializers and Compound Literals." This approach builds upon that specification and assumes that complex numbers are similar to structures with the real and imaginary parts having the names `_real` and `_imag` respectively (although the order is still unimportant).

Yet a third method is an infix operator as in the expression:

```
x ⊗ y
```

Choosing the suitable character(s) for the ⊗ operator is difficult.

By specifying macros to create complex values from real values, each implementation can choose a method most appropriate for that implementation.

7.14.2 (4.14.2) Complex Math Functions

Unless otherwise specified, the domain (or region) and range for all complex math functions is the complex plane with the real and imaginary parts defined for all representable values.

An implementation may set `errno` but is not required to.

Whenever there are multiple mathematical values for a complex function, (multi-valued complex functions) the return value of the function shall be the principal value of the function. Additional rules may be given for a specific function.

For IEEE implementations, the behavior of these functions, if the input contains a NaN, an ∞, or a negative zero, is not specified. (See document X3J11.1/92-061, "Complex and IEEE-754" for more discussion on this topic.)

The `csin` function

Synopsis

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

Description

The `csin` function selects the sine of `z`, where the real part of `z` is regarded as a value in radians.

Returns

The `csin` function returns the sine value.

The `ccos` function

Synopsis

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

Description

The `ccos` function computes the cosine of `z`, where the real part of `z` is regarded as a value in radians.

Returns

The `ccos` function returns the cosine value.

The `cexp` function

Synopsis

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

Description

The `cexp` function computes the exponential function of `z`, where the imaginary part is regarded as a value in radians. An output error occurs if the magnitude of `z` is too large.

Returns

The `cexp` function returns the exponential value.

The `clog` function

Synopsis

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

Description

The `clog` function computes the natural logarithm of `z`. An output error occurs if the argument is zero.

Returns

The **clog** function returns the principal value of the natural logarithm with the imaginary part **w** of the result in the range $-\pi < w <= \pi$. The imaginary part of the result is π only when the real part of the argument is less than zero and the imaginary part of the argument is zero. The sign of the imaginary part of the log is the same as the sign of the imaginary part of **z**.

The cpow function

Synopsis

```
#include <complex.h>
double complex cpow(double complex z1, double complex z2);
```

Description

The **cpow** function computes **z1** raised to the power **z2**. An input error occurs if **z1** is zero and **z2** is not an integral value. An output error occurs if the magnitude of **z1** or **z2** is too large.

Returns

The **cpow** function returns the value of **z1** raised to the power **z2**. For **cpow(0.0,0.0)** the return value is **1.0 + 0.0i**.

The csqrt function

Synopsis

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

Description

The **csqrt** function computes the square root of **z**.

Returns

The **csqrt** function returns the principal value with the real part greater than or equal to zero. If the real part of the result is not zero, then the sign of the imaginary part of the root is the same as the sign of the imaginary part of **z**. If the real part of the result is zero, then the imaginary part is greater than or equal to zero.

The cabs function

Synopsis

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

Description

The **cabs** function computes the magnitude of a double complex number **z**.

Returns

The **cabs** function returns the value of the magnitude of **z**.

The `cimag` function

Synopsis

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

Description

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

Returns

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

The `conj` function

Synopsis

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

Description

The **`conj`** function computes the conjugate of **`z`**.

Returns

If **`z`** has the value of $x+yi$, the conj function returns the value of $x-yi$.

The `creal` function

Synopsis

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

Description

The **`creal`** function selects the real part of **`z`**.

Returns

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

RATIONALE:

Some merging of the header **`<complex.h>`** and the header **`<fp.h>`** will be necessary when this document is merged with document X3J11.1/93-048 "Floating-Point C Extensions."

As long as the complex types are an extension to the standard and not part of the standard, the macro definition for **`complex`** must expand into a name reserved to the implementation. The obvious names are: **`_Complex`** or **`__complex`**. If the extension becomes part of the Standard, this macro definition can be removed.

The definition of the complex operator is given in three macros because different implementors have different needs. No consensus has been achieved within X3J11.1 on a syntax definition for a new operator. A macro allows flexibility in the underlying implementation.

There is no explicit mention of **`errno`** in the descriptions of the complex math functions. Section 4.1.3 of the standard states:

The value of **errno** is zero at program startup, but is never set to zero by any library function. The value of **errno** may be set to nonzero by a library function call whether or not there is an error, provided the use of **errno** is not documented in the description of the function in the standard.

None of the complex math functions are required to use **errno** as an error indicator. Therefore, whether **errno** is changed by any of these functions, is unspecified.

The terms ''input'' and ''output'' errors are used instead of ''domain'' and ''range'' errors so as to not imply that **errno** is set to either **EDOM** or **ERANGE**

The type **double complex** was chosen for the complex functions to correspond to type **double** for the functions in **<math.h>**. More functions can be added to the list if there is sufficient demand.

No specific proposal is made here for handling complex numbers by the **printf** and **scanf** function families. Explicit use of **cimag** and **creal** are sufficient to make these functions perform with complex numbers.