Complex Arithmetic—C9X Edits

WG14/N516 X3J11/95-117 (Draft 12/21/95)

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

This is a proposal for changes to existing parts of the C9X draft document, in order to incorporate CCE. The edits refer to C9X Draft 3. (Subsequent C9X drafts are generally satisfactory for understanding the proposed edits.)

6.1.1 Keywords

In Syntax, add these keywords:

```
complex
imaginary
```

Append to Semantics this paragraph:

The tokens `complex`, and `imaginary` become keywords 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 *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.

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 appropriate to the real type. The low-address and high-address elements of the pair represent the real part and imaginary part, respectively, of the complex value.

The real floating, imaginary, and complex types are collectively called the *floating 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

153

IEEE arithmetic for the complex domain. See [6], [12], [15], and rationale in 6.2.1.5 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`.

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 six types, `float_complex`, `float_imaginary`, `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.]

In the last sentence of the first inserted paragraph above, footnote "imaginary unit" with:

The imaginary unit is a number *i* such that *i* times *i* equals -1.

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

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".

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 Imaginary types

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

6.2.1.6 Real and imaginary

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

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

154

6.2.1.7 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.

6.2.1.8 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.

6.2.1.9 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 or an unsigned 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.

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. The conversion of a real, imaginary, or complex operand yields a type that is real, imaginary, or complex, respectively. The result type is real, imaginary, or complex, as specified for the operator and operand types; a real result has the common real type; each part of an imaginary or complex result has the common real type. This pattern is called the *usual arithmetic conversions*:

First, if either operand has type `long double`, `long double imaginary`, or `long double complex` the other operand is converted to `long double`, `long double imaginary`, or `long double complex` according as that other operand is real, imaginary, or complex, respectively.

Otherwise, if either operand has type `double`, `double imaginary`, or `double complex` the other operand is converted to `double`, `double imaginary`, or `double complex` according as that other operand is real, imaginary, or complex, respectively.

Otherwise, if either operand has type `float`, `float imaginary`, or `float complex` the other operand is converted to `float`, `float imaginary`, or `float complex` according as that other operand is real, imaginary, or complex, respectively.

155

Footnote the third sentence of the inserted text above with:

These conversions do not change the kind—real, imaginary, or complex—of an operand. For example, addition of a `double imaginary` and a `float` entails just the conversion of the `float` operand to `double` (and yields a `double complex` result).

[Automatic conversion of real or imaginary 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,

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

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

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, $6.0 - 0.0i$.

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

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

This also demonstrates the need for imaginary types. Without them, $2.0i$ would have to be represented as $0.0 + 2.0i$, implying that $\text{NaN} + \infty i$ would be the semantically correct result—regardless of conversion rules. And 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.

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.

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, that support 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.

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.]

6.3.5 Multiplicative operators

Append to Semantics the paragraph:

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 real type. If any operand 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 mathematical results of multiplications and divisions involving real, imaginary, and complex operands. *x, y, u,* and *v* 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	(x*u/(u*u + v*v)) + (-x*v/(u*u + v*v))*I	(y*v/(u*u + v*v)) + (y*u/(u*u + v*v))*I	((x*u+y*v)/(u*u+v*v))+ ((y*u-x*v)/(u*u+v*v))*I

For multiplication of two complex operands and division by a complex operand, the usual mathematical formulas shown in the tables do not address quality concerns about undue overflow and underflow (particularly for divide) and undesirable results from infinite operands. Also, certain schemes to handle the over/underflow problems cause surprising roundoff errors. For implementation guidance, see Annex X.11.1.1, [6], and [9].]

157

6.3.6 Additive operators

Append to Semantics the paragraph:

If one operand has real type and the other operand has imaginary type, then the result has complex type. If both operands have imaginary type, then the result has imaginary type. If any operand 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 at translation time, without floating-point arithmetic. Examples include *y * I*, *x + v * I*, and *I * I*.]

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 (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.]

158

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 imaginary and complex types, as well as 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 an imaginary or 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
imaginary
```

In Constraints, add to the bullet items:

— `float complex`
— `float imaginary`
— `double complex`
— `double imaginary`
— `long double complex`, or `long complex`
— `long double imaginary`, or `long imaginary`

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`".