Comments on Proposed ANSI C Standard

David Hough - dhough@sun.com

ABSTRACT

The proposed C standard suffers numerical shortcomings - many inherited from its precursors - in areas of interest to providers of portable mathematical software. I comment in detail upon the following aspects of the proposed standard:

Comment #1, Section 1.1:	anticipate supplemental standards
Comment #2, Section 1.1:	manifest the best way
Comment #3, Section 2.2.4.2:	exclude implementations explicitly
Comment #4, Section 2.2.4.2:	use "significand"
Comment #5, Section 3.1.3.1:	bound rounding errors explicitly and uniformly
Comment #6, Section 3.2.1.4:	round conversions between floating types
Comment #7, Section 3.2.1.5:	forbid implicit narrowing conversions
Comment #8, Section 3.3.3:	add power operator for integral exponents, or square
Comment #9, Section 3.3.4:	emphasize rounding forced
Comment #10, Section 3.4:	defer constant expressions with side effects
Comment #11, Section 3.5.6:	encourage non-zero initialization
Comment #12, Section 4.7:	SIGFPE means floating point
Comment #13, Section 4.8:	variable argument lists are expensive
Comment #14, Section 4.9.6:	printf/scanf duality for non-model numbers
Comment #15, Section 4.9.6.1:	display signed zero with printf %+
Comment #16, Section 4.9.6.1:	distinguish exact zero with printf %f
Comment #17, Section 4.9.6.1:	provide useful printf %#g
Comment #18, Section 4.9.6.2:	scanf requires more than minimal ungetc
Comment #19, Section 4.10.1.4:	strtod/atof are mathematical functions
Comment #20, Section 4.10.2:	require two random number generators
Comment #21, Section 2.2.4.2:	<float.h> has too many names, not enough information
Comment #22, Section 2.2.4.2:	long double > minimal float
Comment #23, Section 3.1.3.1:	compiler conversion same as run-time
Comment #24, Section 3.3.2.2:	no implicit declarations
Comment #25, Section 3.5.4.2:	fix arrays
Comment #26, Section 3.7.1:	standardize Fortran-77 interface
Comment #27, Section 3.8.8:	predefine generalized precision macros
Comment #28, Section 4.5:	specific mathematical library functions
Comment #29, Section 4.13.4:	standard functions predefine generic operators
Comment #30, Section 4.5.1:	make numerical exception handling uniform
Appendix #1:	A Proposal for Conformant Arrays
Appendix #2:	A Proposal for <float.h>
Appendix #3:	Why does traditional C treat float the way it does?

Preface

The following comments were submitted to X3J11 as part of its second public review period. X3J11's subsequent **Summaries** of the issues and **Responses** are included. The Summaries and Responses are each exactly one paragraph, and may thereby be distinguished from my commentary

before and after. Note that each Response has a one-line generic response followed by one or more lines of specific response. These are not always self-consistent.

The third public review period, based upon the document number X3J11/88-090, dated 13 May 1988, concludes 1 September 1988.

The following comments are based upon *Draft Proposed American National Standard for Information Systems - Programming Language C*, document number X3J11/88-002, dated 11 January 1988, and its accompanying Rationale. The comments are personal opinions of the author, and should neither be construed as wholly original nor as representing the position of any organization or other person. A number of individuals helped formulate and clarify them; some of their names are listed here and at the end. The following, while not necessarily agreeing in every detail, have expressed agreement with the main points:

Greg Astfalk	cleast!astfalk
Larry Breed	ibmpa!lmb
D. Burton	phtoint%bnandp10.bitnet
W. J. Cody	cody@anl-mcs.arpa
Iain Johnstone	iainj@playfair.stanford.edu
W. Kahan	University of California, Berkeley
Zhishun Alex Liu	zliu%hobbes@berkeley.edu
David Mendel	mendel@playfair.stanford.edu
Jim Meyering	meyering@cs.utexas.edu
K-C Ng	kcng@sun.com
Gene Spafford	spaf@purdue.edu
Philippe Toint	phtoint%bnandp10.bitnet
Stein Wallace	wallace%nocmi.bitnet

Some people agreed with the intent of the wider-ranging proposals but felt they were offered too late in the standardization effort. To such arguments the best response is that it's never too late to fix problems because fixing is always cheaper sooner than later, especially in standardization. Many X3J11 members are anxious to obtain early publication of the standard, but due to several recent controversial changes and some aspects of the Draft that might be controversial if their ramifications were more widely understood, there is a good chance that yet another complete public review cycle will be required.

Introduction

C was not particularly designed to facilitate numerical computation. In recent years it has come to be increasingly used for implementing portable systems and applications, including numerical ones. This is more a tribute to the good judgment embodied in some other aspects of C's design than to its numerical facilities: it is easier to get a usable C compiler and library working than a comparable Fortran compiler and library. Examples of such applications are the SPICE 3B1 circuit simulation package, which is far more flexible and maintainable than its Fortran predecessors, and Alex Liu's elementary function test programs, which would have been far more difficult to implement in Fortran.

In its drafts the ANSI committee has removed some of traditional C's numerical weaknesses, such as requiring double-precision expression evaluation and parameter passing, and overspecifying error response in the elementary transcendental function library. With a little more effort the remaining numerical stumbling blocks could be removed and C would be as convenient for the numerical parts of applications as for the other parts.

Comments follow, approximately in order of increasing significance, complexity, and priority. Most of what follows is more a critique of existing C implementations than of the X3J11 committee's work. For the simpler issues, specific **Recommendations** and wording are provided. In more complicated cases, the principles are exposed rather than the details; details can be worked out for proposals accepted in principle. Sometimes, an **Alternate Recommendation** is provided should the primary recommendation be deemed too great a step to take.

In much of what follows, "infinity" refers to a floating-point representation that is intended to act like mathematical ∞, such as is found on CDC and IEEE implementations. Similarly "NaN" refers to any kind of non-numeric floating-point representation, including CDC Indefinite, VAX Reserved Operand, and IEEE NaN.

Comment #1, Section 1.1: anticipate supplemental standards

Certain aspects of the definition of C should be deferred until X3J3 completes its work on Fortran 8x. These aspects include

* the environmental inquiries currently defined in 2.2.4.2 for <float.h>;
* a generalization of the Fortran-77 interface proposed for 3.7.1.

Since it's desirable in most cases for C to follow rather than lead Fortran in these areas, C standardization should be deferred until the Fortran work has been tested in practice.

Many C implementations for personal computers, workstations, and superminicomputers are built upon the IEEE 754 standard for binary floating-point arithmetic. Certain aspects of the C environment could well be standardized among such implementations, without impacting the many non-IEEE C implementations:

* Definition of the proposed sig_detail on IEEE implementations;
* Definition of operators or procedures for IEEE modes and status;
* Definition of syntax for infinities and NaNs.

The X3J11 Draft should avoid prejudicing such later work.

Most X3J11 and X3J3 members suffer various degrees of standardization fatigue, so it would be appropriate for another body to develop an informal standard a couple of years after Fortran 8x is finalized; that informal standard might be incorporated into a subsequent revision of C. An IEEE Trial-Use Standard or Recommended Practice, for instance, may be developed more quickly than a formal ANSI Standard.

X3J11 Summary: Anticipate supplemental C floating-point standards.

X3J11 Response: The Standard must accommodate a variety of environments. The X3J11 Committee tries to accommodate existing standards and existing implementations. It is far more difficult to anticipate the future directions of other standardization efforts. The Committee believes the floating-point model provides useful information for a broad class of floating-point applications and is part of our charter.

Comment #2, Section 1.1: manifest the best way

Section 1.1 of the Rationale mentions certain aspects of C design revered by tradition. Some of these could be restated as "Make the best way manifest", meaning that it should be clear what the best way to perform an intended operation is. This principle has not been faithfully followed in the past: which is the best way to increment a: ++a , a++ , a+=1 , or a=a+1 ? If the answer is different on different compilers, then how should portable efficient programs be coded?

Several of the proposals which follow are attempts to make the best way to perform certain operations manifest by making them defined language features, so that any poor performance may be justifiably blamed on the compiler and library rather than on the programmer.

X3J11 Summary: Manifest the "best way".

X3J11 Response: This was not considered an issue requiring specific action. This particular criticism was non-specific, so no action is possible.

Comment #3, Section 2.2.4.2: exclude implementations explicitly

The floating-point model in section 2.2.4.2 is a signed-magnitude one and implicitly excludes certain other types of floating-point implementations that have existed in the past and might be considered again. If this exclusion is intentional it should be explicit; if accidental then substantial modifications

are necessary in this section. Note that the Draft unashamedly mandates binary integral types.

Logarithmic floating point has been implemented in embedded systems at Lockheed and elsewhere, and may offer attractive performance again in the future, at least for low precision. The entire model of representable numbers is quite different for such floating point, and section 2.2.4.2 would be substantially recast to encompass it.

Previous one or two's-complement floating-point implementations shared with complemented integer arithmetic the unhappy consequence that negation is not a trivial operation and may overflow or otherwise misbehave. Section 2.2.4.2 would require some modifications to accommodate such arithmetic. Section 3.2.1.3 specifies that "the fractional part is discarded" to convert from floating to integral type. Unless complemented floating-point representations are excluded, the Draft's intent won't be clear without adding "by rounding toward zero."

Recommendation: In 2.2.4.2, add "The foregoing signed-magnitude model intentionally excludes one's and two's-complement floating-point representations and logarithmic floating-point representations."

X3J11 Summary: The floating-point model implicitly excludes certain implementations.

X3J11 Response: Changes have been made along the lines you suggested. A footnote has been added to 2.2.4.2 stating that the floating-point model requires a sign-magnitude model.

Comment #4, Section 2.2.4.2 and 3.1.3.1: use "significand"

The term *significand* was adopted by the IEEE floating-point committees to designate the part of a floating-point number that contains its significant digits. Significand is a better term than either "mantissa", which refers to the fractional part of a logarithm, or "value part", which implies that the exponent doesn't contribute to the value of a number.

Recommendation: Replace "mantissa" and "value part" with "significand" throughout the Draft and Rationale.

X3J11 Summary: Replace "mantissa" and "value part" with "significand".

X3J11 Response: The Committee believes this is clear enough as is. We believe that the use of "mantissa" is synonymous with "significand" and the use of "value part" is unambiguous.

Comment #5, Section 3.1.3.1 and A.6.3.5: bound rounding errors explicitly and uniformly

A.6.3.5 mentions that rounding directions are implementation-defined for casts from integers to floating-point types and between floating-point types, as are "the properties of floating-point arithmetic". But in several places, such as 3.1.3.1 and 3.2.1.3, the Draft defines an acceptable floating-point rounding method for an unrepresentable unrounded value as choosing one or the other of its two nearest representable neighbors.

If this is a laudable effort to raise the quality of floating-point implementations, excluding almost all existing C implementations, that should be stated explicitly. If unintentional the committee must at least relax the accuracy requirements in 3.1.3.1 governing decimal-to-binary conversion in the compiler.

Recommendation: In the preface to chapter 3, add a new paragraph:

> *Acceptable floating-point rounding:* When an exact floating-point numerical value, whose magnitude does not exceed that of the largest finite representable number, is to be rounded to a representable floating-point value, then that representable floating-point value shall be chosen from the two representable values nearest the exact value in an implementation-defined manner. Consequently, if the exact value is representable, then the rounded value shall be that representable value. This *acceptable rounding rule* also encompasses floating-point underflow but not floating-point overflow. Acceptable floating-point roundings govern:

1 conversions between floating-point types and from integral types to floating-point types, but not conversions from floating-point types to integral types;

2 conversions between ASCII representations and internal floating-point formats by strtod(), scanf(), and printf(), or as constants in source code;

3 operators +–*/ including constant expressions evaluated at compile time;

4 functions fabs(), fmod(), frexp(), ldexp(), modf(), sqrt(), ceil(), and floor().

Note that of these functions, the correct computed value of fabs(), fmod(), frexp(), modf(), ceil(), and floor() is alway representable if numeric, and ldexp() only rounds in the event of underflow and overflow.

Alternate Recommendation: Permit conversions from floating-point formats to narrower floating-point formats, and from integral formats to floating-point formats, to be performed with the same sorts of rounding permitted other arithmetic operations, namely those provided by the underlying system. Document the digits of accuracy of input and output conversion in <float.h> as suggested below in Appendix #2.

X3J11 Summary: Bound rounding errors explicitly and uniformly.

X3J11 Response: The Committee believes that this is clear enough as is. We believe that the rounding behavior specified in the Standard is correct.

Comment #6, Section 3.2.1.4: round conversions between floating types

The C Draft requires that conversion of a floating-point value to fit in a floating-point format of less precision or exponent range be accomplished by an acceptable rounding operation as defined above; acceptable roundings include rounding toward zero. This is all consistent with IEEE arithmetic. But the Rationale still asserts that conversions to floating-point format by rounding to nearest would be inefficient, because rounding modes would have to be frequently changed, since rounding toward zero is required for the conversion of floating-point values to integer formats. This assertion, reflecting an earlier X3J11 draft, is incorrect and misleading and should be removed.

Actually the efficiency of conversion from floating-point formats to integer formats is not a major factor in overall performance of most realistic applications. Even so, better IEEE implementations recognize that while dynamic rounding modes are appropriate for most floating-point operations, those producing an integral value in integer format (int cast) or an integral value in floating-point format (floor()) should not be subject to variation at run time. The implementations provide an additional operation beyond those those mandated by the IEEE standard. That additional operation (fintrz on Motorola MC68881) converts floating-point values to integers, rounding toward zero regardless of the current IEEE rounding mode that governs other floating-point operations. So there need be no efficiency lost by requiring, as the Draft does, conversion to integer to round toward zero, even if that differs from the rounding mode for operations with floating-point results.

Recommendation: Change the rationale to read:

> The Standard, unlike the Base Document, does not require rounding to nearest in the double to float conversion. Conversions between floating-point types must be rounded according to the same acceptable rounding rules applicable to other roundings of floating-point results.

X3J11 Summary: Remove an incorrect assertion about round conversions between floating types.

X3J11 Response: The Committee has voted against this idea. There are existing implementations where rounding to nearest is inefficient. (The PDP-11 and 8087 families behave this way.)

Comment #7, Section 3.2.1.5 and A.5: forbid implicit narrowing conversions

Appendix A.5 mentions a warning for implicit conversions to narrower formats, presumably meaning conversions which might be inexact or otherwise exceptional, including conversions between floating-point formats, conversions from floating-point to integer formats, and some cases of conversion from integer to floating-point format. Better to forbid such implicit conversions outright since the implicit roundings so often lead to elusive bugs. Correct existing code that is careless in this respect is

easily remedied by explicit casts.

Note that code like

```
float f ;

f = 1.0 ;
```

or

```
f = 1 ;
```

results in an implicit narrowing conversion, but is unobjectionable since there is no possibility of roundoff error or other side effect. Such code is best thought of as a constant expression, acceptable for compile-time evaluation, according to rules proposed above.

Recommendation: In 3.2.1.5, add:

Conversions, by assignment or by the usual arithmetic conversions, of values from one integral or floating-point type to another integral or floating-point type are not permitted except:

1 When the conversion is invoked by an explicit cast operator;

2 When the destination type can represent all values of the source type;

3 When the destination type can represent the known source type value to be converted, as in the case of conversion of constants or constant expressions such as float f = 1.0, g = 1<<16 .

X3J11 Summary: Forbid implicit narrowing conversions.

X3J11 Response: The Committee has voted against this idea. We believe that the practice of allowing implicit narrowing conversions is widespread existing practice. Explicit prohibition of this practice would break too much existing code.

What about lint? A common response to complaints about implicit conversions and other questionable programming practices is "lint catches those" - so the compiler need not. A common counter-response is "lint's not part of the Draft". Both viewpoints might be accommodated by affirming or denying, in the Rationale, X3J11's presumed intent that compilation systems that include a separate checking program such as lint need not perform redundant checks in the compiler itself, and that conversely, systems that don't provide such a checking program must do it all in the compiler.

X3J11 Summary: Clarify whether "lint" checking may be separate from compilation.

X3J11 Response: Quality of implementation is beyond the scope of the Standard. An implementation is at liberty to provide warnings about implicit narrowing.

Comment #8, Section 3.3.3: add power operator for integral exponents, or square

C does not provide a power or exponentiation operator, like Pascal (which however provides a squaring operator), but unlike Fortran ** or Basic ^ . C's original system programming motivation does not require such an operator, and C's general approach is that operations that can not be implemented in a few machine instructions should be invoked as functions. Based on Fortran experience, this is excellent design; it discourages the common Fortran codings "x**0.5" and "x**2.0". Conversely, omitting an easy way to say "x**2" for complicated expressions x is a C oversight that adds gratuitous complexity to code.

The complexity of general exponentiation to a floating-point power is sufficiently attested by the dropping of such an operator from the final MC68881 architecture due to lack of microcode space in the initial implementation. Therefore I don't advocate that C add an *operator* notation for floating-point powers (although later I advocate that functions like pow() be made an operator in *functional* notation). I do advocate that an operator notation for exponentiation to integral powers be added. That some convenient syntax be standardized is more important than the specific syntax, and X3J11 is best qualified to choose from various alternative candidates such as *^, **, ^^, %%, ->, etc. I'll use *^ in the following, although anybody who has written x**y or x ** y in old C code, intending x * *y, has probably done some other things that won't pass an ANSI C compiler. Even though ANSI C could be defined to accept x**y in its old meaning as well as x**y in its new since y must be a pointer in the first case and

an integral expression in the second, I doubt that's worthwhile.

Although the current Draft permits an implementation to optimize its interpretation of pow() to treat integer exponents differently, would-be implementers of portable efficient mathematical software libraries might be disinclined to rely on such local optimizations, providing their own floating *^ integral functions instead. Thus instead of concentrating their effort upon functions which should never be part of a language standard, they divert their energies to providing facilities that languages could have easily provided.

To best correspond to the usage of mathematics and other computer languages, the precedence of *^ must be greater than that of a multiplicative or additive operator or unary –. That seems to imply that *^ has "postfix operator" precedence.

Recommendation: Modify 3.3.3 and add a new section 3.3.3.5 defining power operators.

Syntax: Add to the definitions for postfix-expression:

```
power-expression:
        postfix-expression
        postfix-expression *^ cast-expression

unary-expression:
        power-expression
        ++ unary-expression
        ...
```

Constraints: In b *^ n, the base expression b must be an integral or floating expression, but the exponent expression n must be an integral expression.

Semantics: If n == 0, then the result is 1 for any b, including 0 or NaN, cast to the type of b. If n > 0, then the result is the usual: the value of b is multiplied together n–1 times. If n < 0, then the result in the absence of round-off, overflow, or underflow, would be 1/(b *^ (–n)), which is 0 if b is an integral expression >= 2. Unlike most other operators, but like b << n, the "usual arithmetic conversions" are not made to b and n; the type of the result is always that of b. "x *^ y *^ z" has its usual mathematical meaning "x *^ (y *^ z)".

Section 3.3.16: Add an assignment operator *^=.

The Rationale should encourage compilers to be clever when n is constant –1, 0, 1, or 2, but not so clever with other n that the intermediate results overflow or underflow unnecessarily, as in the attempt to compute b *^ 7 as (b *^ 8)/b. When n <= –2 for floating-point b, avoid roundoff by computing 1/(b *^ –n) and reciprocating, unless the effects of overflow or underflow are best mitigated by computing (1/b) *^ n instead. The Rationale should also encourage compilers to compute, wherever possible, "x * b *^ n" by a shift instruction when b is the base of integer arithmetic, and by ldexp(x,n) when b is the base of floating-point arithmetic.

Although it would be convenient to allow b *^ -n to be undefined for integer b and n, so that 2 *^ m or 16 *^ m could be implemented by shifts 1 << m or 1 << 4*m which are also undefined for m < 0, that would mean that programs could not rely on limited identities like b *^ (m-n) = b *^ m / b *^ n, *true for non-negative m and n.*

Alternative Recommendation: Add a new section 3.3.3.5 defining a new unary operator square in functional notation that computes the square of the value of its argument without re-evaluating its argument.

This alternative takes care of the most common case of *^, making manifest the fastest way to square an expression, without adding a new operator syntax.

X3J11 Summary: Add an integral power operator.

X3J11 Response: The Committee has reaffirmed this decision on more than one occasion. Proposals to add a power operator have been rejected many times. It is possible to treat the function pow as a reserved function and thereby perform the operation in-line. This is especially useful when the

exponent has integral type.

Comment #9, Section 3.3.4: emphasize rounding forced

In many traditional C implementations, it is not easy to force a rounding from double to float to occur; the statements

```
register double e;
register float f;

f = (float) e;
```

won't cause any rounding to occur, confounding any program attempting to determine properties of floating-point arithmetic or attempting to purify a test argument to a float function. To force the rounding, statements like

```
register double e;
register float f;
float g;

g = (float) e;
f = *(&g);
```

OR

```
register double e;
register float f;
volatile float g;

g = (float) e;
f = g;
```

are needed, at the cost of gratuitous memory operations. "Gratuitous memory operations" don't sound like much in a personal computer without floating-point hardware, but in high-performance environments, stores and loads are often more costly than floating-point operations.

The Draft appears to implicitly constrain all implementations to respect casts to the extent of forcing the value cast to fit in the cast type. Here "casts" include explicit casts such as f = (float) d and implicit casts in assignments (f = d) and function value returns (return d).

This constraint is laudable but perhaps not widely recognized since many existing implementations follow the tradition and therefore would not be standard-conforming. If this is a potentially contentious issue the sooner it is exposed and dealt with, the better.

Recommendation: In the Rationale, state explicitly that existing implementations which do not always round to fit explicit and implicit casts are NOT conforming.

Function results in long double registers: Implementations may readily pass all floating-point parameters - float, double, and long double - in long double format by virtue of the "as if" principle. Furthermore float, double, and long double function results may be returned in long double containers, for the same reason, *as long as float and double function results have been rounded to their respective precisions prior to promoting the rounded result back to long double form.* This rounding properly occurs in the called function.

X3J11 Summary: Emphasize that casts force rounding.

X3J11 Response: Changes have been made along the lines you suggested. Words have been added to the Rationale to emphasize that casts require conversions.

Comment #10, Section 3.4: defer constant expressions with side effects

The C standard committee recognized some of the problems associated with constant expression evaluations involving floating point, when it mandated that compile-time constant expression evaluation must be at least as accurate as the run-time environment. Even so there are occasions when, for clarity, one would like to write a constant expression while deferring its evaluation to run time. Thus

```
z = -1.0/0.0 ;
```

may be intended to generate an infinity and division-by-zero exception at run time (perhaps within an implementation of log). The exception might be lost if the expression were evaluated at compile time; some compilers might refuse to compile that expression until it was suitably disguised. Such disguises do nothing to improve code readability; the standard should preclude compile-time expression evaluation except in cases free of side effects or when side effects were explicitly ignored. In this context compile-time expression evaluation includes statements like

```
double z = 1.0e999 ;
```

and

```
a += 1.0/3.0;
```

which generate exceptions on some run-time systems. Thus expressions like those above would not normally be evaluated at compile time.

Most integer expressions involving only small constants could be evaluated at compile time under any circumstances, as before. Otherwise expressions may only be evaluated at run time and subject to the same sequencing constraints as other executable statements so that, for instance, evaluation of all constant expressions at the beginning of program or function execution is impermissible.

Recommendation: Modify 3.4 Description to read:

In general, *constant expressions* must be evaluated at run time rather than compile time. There are three exceptions:

1 When the constant expression initializes a variable with the **static** attribute;

2 When the language syntax requires an integral constant (footnote 40);

3 When the constant expression may be evaluated at compile time without any rounding error or other numerical exception, and the result is exactly representable in the run-time environment (this includes most expressions involving small integer constants);

Even in these cases, a warning must be issued if the constant expression can't be evaluated exactly.

The "as-if" rule comes to the aid of many implementations, of course; on systems in which floating-point expression evaluation produces no side effects, compile-time evaluation is indistinguishable from run-time.

X3J11 Summary: Defer *constant expressions* with side effects.

X3J11 Response: Quality of implementation is beyond the scope of the Standard. An implementation is at liberty to defer floating-point constant folding until execution time.

Comment #11, Section 3.5.6: encourage non-zero initialization

Like implicit function typing, implicit initialization has been a source of many C bugs. Neophytes, readily, and experienced programmers, sometimes, confuse the situations in which variables may be depended on to be initialized to zero with those where they may not, or more commonly don't even think about initialization.

And automatic initialization to zero, a natural integer and floating-point value, is not nearly as effective at catching logical errors as initializing memory to 0xffffffff, for instance, on a 2's complement IEEE machine.

Recommendation: In section 3.9, classify programs that depend on implicit initialization to zero as obsolete. In section A.5, encourage compilers to detect as warnings as many cases as possible of use before definition. As a common extension, encourage compilers to provide means by which use of obsolete features generates an error message.

X3J11 Summary: Deprecate implicit zero initialization.

X3J11 Response: This proposal conflicts with too much prior art. This would break too much existing code and is considered useful by many programmers.

Comment #12, Section 4.7 and A.6.5.14: SIGFPE means floating point

SIGFPE implementations often encompass many signals besides its namesake floating-point exceptions. The handling possibilities for these other types of signals vary greatly and are mostly machine-dependent. But the types of floating-point exceptions are comparatively uniform: operand exceptions such as reserved operand, invalid operation, or division by zero, and result exceptions such as floating-point overflow, underflow, significance loss, inexact, or integer overflow on conversion from a floating-point format. The C standard would best serve the cause of portability by specifying that only floating-point exceptions of the general types mentioned may generate SIGFPE. Such a requirement does not impact implementations that don't generate any signals or don't generate SIGFPE, but does make it more likely that work like that described by David Barnett (david@lll-lcc.arpa) on "A Portable Floating-Point Environment" will be widely available via portable SIGFPE handlers without disrupting other types of exceptions that properly arise elsewhere.

Recommendation: Change 4.7 to read:

> SIGFPE a floating-point exception arising from a floating-point instruction, including operand exceptions such as reserved operand, invalid operation, or division by zero, and result exceptions such as floating-point overflow, underflow, significance loss, inexact, or integer overflow on conversion from a floating-point format. Exceptions arising from non-floating-point instructions, such as integer overflow and integer division by zero, and non-numerical exceptions arising from floating-point instructions such as illegal instruction or address, shall NOT generate SIGFPE.
>
> SIGARITH an arithmetic exception arising from an operation producing integral results from integral operands, such as integer overflow or integer division by zero. Non-numerical exceptions such as illegal instruction or address shall NOT generate SIGARITH.

X3J11 Summary: Reserve **SIGFPE** for floating-point exceptions only.

X3J11 Response: This proposal conflicts with too much prior art. Many implementations raise floating-point exceptions for a variety of reasons. Exception handling is very implementation dependent. The Committee would like to make signals more portable but the variety of architectures and implementations makes that task very difficult.

Even if only genuine floating-point exceptions generated SIGFPE, most signal handlers would probably handle some differently from others. The BSD method for transmitting some detail about the signal is to pass an additional argument (int code) although a pointer to a struct whose details are machine-dependent would be a better choice. SIG details aren't suitable for standardization but the method of transmitting them - by an extra parameter in the signal handler calling sequence - is. Despite a comment in A.6.5.14, adding extra parameters does not seem to be permitted by the Draft. The intent is to allow subsequent standardization of struct sig_detail on IEEE machines.

Recommendation: Add to 4.7:

> <signal.h> defines an implementation-dependent struct sig_detail, possibly trivial.
>
> Add to 4.7.1:
>
> When a signal occurs, the signal handler is invoked as (*func)(int sig,struct sig_detail *psd). When setting up input parameters for (*func), an implementation may pass a null psd if there are no details to report.

Add to 4.7.2.1:

int raise(int sig, struct sig_detail *psd);
The raise function sends the signal sig to the executing program, accompanied by signal details in *psd, which may be null if there are no details.

X3J11 Summary: Add an implementation-dependent argument to signal handlers.

X3J11 Response: The Standard reflects widespread existing practice in this regard. There is no prior art for this feature (that is, not for a technically correct approach). Adding this feature would break too much existing code. The Committee attempts to codify existing practice whenever possible.

Some implementations will never detect any floating-point exceptions under any circumstances. Then SIG_ERR and errno indications could be returned by signal() rather than silently accepting an attempt to specify a SIGFPE handler which can only be invoked by raise(). Alternatively, require <signal.h> to contain #define can_SIG... for each X3J11-standardized SIG that an implementation can generate by means other than raise():

Recommendation: Add to 4.7:

For each standardized exception such as SIGABRT, an implementation's <signal.h> also defines macros #define can_SIGABRT if that signal can arise by other means than raise().

X3J11 Summary: Need a way to determine which signals may be produced by means other than raise.

X3J11 Response: A specific proposal is needed before action can be taken. No prior art exists for this feature. Many aspects of the signal function are implementation defined. This is due to the wide variety of implementations in existence, each of which has unique requirements.

Comment #13, Section 4.8: variable argument lists are expensive

Variable argument lists impose a burden on the Draft far in excess of their value. The principal reason they must be standardized is so the printf and scanf families of functions can be specified as part of the C run-time library rather than as part of the language. Furthermore the necessity of providing for variable argument lists constrains system designers far out of proportion to the benefit; although the Draft does not proscribe different calling conventions for variable-argument and fixed-argument functions, most implementations take the conservative course of providing a common calling sequence, for backward compatibility if nothing else. A consequence of this is that, for instance, floating-point parameters and integer parameters will be passed in the same registers or on the same stack whether that is most efficient or not.

It may not be widely recognized yet that despite X3J11's efforts to provide backward compatibility and portability, *the Draft requires every ANSI C usage of printf or scanf to have in scope an ellipsized variable-argument function prototype.* Many otherwise legitimate existing programs that don't #include <stdio.h> won't work on an implementation which exploits the freedom granted by the Draft to pass variable argument lists differently from fixed lists. Consequently many new implementations will constrain themselves so that these older programs continue to work.

Later I recommend that *all* function invocations be required to have a prototype in scope. If that recommendation were adopted then the problems of backward compatibility of printf and scanf would be much less pressing.

X3J11 Summary: Variable argument lists are expensive.

X3J11 Response: Quality of implementation is beyond the scope of the Standard. An implementation is at liberty to implement **printf** in such a way that **printf** will behave correctly without a prototype in scope (as you note). However, a simple compile-time switch could provide a faster calling sequence for strict ANSI C programs. We also note that many applications make use of **printf**-like error logging functions.

Comment #14, Section 4.9.6 and 4.10.1.4: printf/scanf duality for non-model numbers

The values of ...MIN_DIGITS and ...MAX_DIGITS proposed for <float.h> below constitute a specification on printf and scanf; under appropriate conditions, model numbers are guaranteed to reproduce their values under input|output and output|input sequences.

Infinities and NaNs should also be printable with printf and scannable with scanf and strtod in the same way as numeric tokens. Ideally the C standard should specify the format of acceptable character representations for floating-point infinity and NaN. At a minimum, all implementations should recognize "infinity" on input, and convert it to the largest representable magnitude in the target format; all implementations should recognize "nan" on input and convert it to a non-numeric symbol if one exists, otherwise indicating an error different from that corresponding to a totally unrecognized character string. "infinity" and "nan" should be recognized in any combination of upper and lower case.

Given the reluctance of the IEEE 754 committee to prematurely commit to such specific representations, the C committee might impose a requirement less demanding, but correspondingly less beneficial: non-model numbers and non-numbers, when printf'ed in an output format that accommodates ...MAX_DIGITS significant digits for model numbers, should be readable by scanf and strtod with an input format that reproduces model numbers. Such an output|input sequence should map model numbers to themselves, non-model numbers to themselves, and non-numbers to non-numbers. That guarantees that output formats sufficient to preserve numeric information also preserve non-numeric identity.

The Draft may be interpretable as already requiring the capability described in the preceding paragraph.

Recommendation: Add to the Rationale:

> An implementation that includes infinities or NaNs shall printf them in the usual numeric format output fields with appropriate character strings that distinguish them from finite numbers. Those character strings must also be readable by strtod and by scanf in the usual numeric format input fields and converted into the appropriate internal representation of infinity or NaN respectively.

X3J11 Summary: Require **printf/scanf** duality for non-model numbers.

X3J11 Response: Quality of implementation is beyond the scope of the Standard. An implementation can assign meaningful semantics to printing and scanning non-model numbers. We felt it would be an undue burden to require all implementations to support non-model numbers.

Comment #15, Section 4.9.6.1: display signed zero with printf %+

Implementations where the sign of zero is meaningful require a method of forcing its display in printed output. The simplest would be to amplify the definition of the + modifier of printf output to explicitly state that it causes the sign of zero to be displayed.

Recommendation: Add to Rationale:

> Implementations in which the sign of zero is significant shall always display that sign when the + modifier is included in a print specification. Implementations in which the zero has no sign or zero's sign bit has no significance shall always display a + sign.

X3J11 Summary: Display signed zero with **printf %+**.

X3J11 Response: Quality of implementation is beyond the scope of the Standard. There are ANSI standards that forbid displaying a negative zero. The FORTRAN-77 standard does not allow displaying negative zero and currently the proposed FORTRAN-8x standard does not allow this. An implementation is at liberty to implement this feature.

Comment #16, Section 4.9.6.1: distinguish exact zero with printf %f

In debugging problems that are costly or difficult to reproduce, it is helpful to be able to distinguish, without changing and rerunning the program, exact zeros from small non-zero numbers that only display zero digits in %f or %g format. To this end the specification for %f format should state that the

result for exact zero is 0 followed by n+1 spaces, where n is the precision specification, instead of a decimal point and n zeros. The intent of the alternative format %#f would be better preserved, however, if it caused even exact zero to be printed with an explicit decimal and trailing zeros.

Recommendation: Add to 4.9.6.1 description of f format:

> If the argument is exactly zero and the # alternate form is not specified, then blanks are used in lieu of the decimal-point character and its trailing zeros.

X3J11 Summary: Display exact zero with **printf %f.**

X3J11 Response: Quality of implementation is beyond the scope of the Standard. The Committee was reluctant to add another special case (there are enough special cases already). An implementation is at liberty to assign these semantics to **printf.**

Comment #17, Section 4.9.6.1: provide useful printf %#g

Printf supplies %g and %#g conversions. The difference between them is not very interesting. What would be more useful is a %#g that exploits available space better than normal %g.

Recommendation: Replace the # description for g and G as follows:

> Printf with a %#g specification ignores any precision specification and bases formatting decisions solely on the width specification w. If the operand is an exact integer that can be printed within a field of width w, then it is printed without a decimal point or trailing zeros. Otherwise the operand is printed in the %f format that displays the maximum number of significant figures, with the decimal point anywhere within the field, unless 1) the operand is large enough that there would be no room for the point within the field, or 2) the operand is small enough that it would display more significant figures in a %e format. In either of these last two cases a %e format is used.

Except when the value can be represented exactly as an integer, 1) the full field width w is used, and 2) a numerical value is never printed without a decimal point. More than w characters are used only if w is so small and the operand so extreme that a %e representation with one significant figure would exceed w characters.

X3J11 Summary: Provide useful **printf %#g.**

X3J11 Response: A specific proposal is needed before action can be taken. The Committee felt there was no prior art for this proposal.

Comment #18, Section 4.9.6.2: scanf requires more than minimal ungetc

The current Draft finally answers most of the questions about the interaction of ungetc() and scanf(). The Draft seems to imply:

> When scanf() processes a *correct* format conversion and reads beyond it, that first character beyond the formatted field is guaranteed to have been pushed back to the input stream. When scanf() processes an *incorrect* format conversion, at most one input character is guaranteed to have been pushed back by scanf() - so that any previous characters read and subsequently rejected by scanf may have been lost, as in the case of attempting to scan "-.x" with %f. In either case, after a return from scanf(), ungetc() must be able to accept at least one more character of pushback. Thus if the ungetc() implementation is minimal then scanf() must be implemented with some more powerful pushback mechanism.

Many existing sscanf implementations implement ungetc by writing back characters to the input string. This causes problems if that string is in read-only memory. The Draft appears to reject such implementations by declaring "const char *s" and the Rationale should emphasize the point in English.

X3J11 Summary: Emphasize that **fscanf** requires more than minimal **ungetc.**

X3J11 Response: The Committee has voted against this idea. Your description of the **scanf/ungetc** semantics are correct. An implementation is at liberty to implement writable string literals. A maximally portable program cannot rely on strings being writable. If strings literals are writable then **sscanf** could be implemented in such a way that no program could tell that a character

was actually being written over itself.

Comment #19, Section 4.10.1.4: strtod/atof are mathematical functions

From the point of view of exception handling, strtod() and atof() are like the mathematical library functions discussed later. Thus if no error handling is specified, the numerical results of underflow or overflow or invalidly-formed input strings should be undefined. If error handling is specified, the treatment of underflow and overflow should be like that of multiplication, and the treatment of an invalid input string should be like that of 0.0/0.0.

Recommendation: In 4.10.1.4, remove errno specification for strtod().

X3J11 Summary: Remove the **errno** specification for **strtod.**

X3J11 Response: The Standard reflects widespread existing practice in this regard. We sympathize with your desire to remove **errno, strtod, strtol, and stroul,** but there is too much existing practice. It should be noted that these functions are different from the math functions in that they are not required to execute as a single operation without visible exception.

There must be a separate implementation strtodl() for long double and there should be a separate strtodf() for efficiency. Unfortunately these names exceed six characters; strtof is unused but not strtol! Speed of base conversion is relatively important in applications where input/output dominates computation.

Recommendation: In 4.10.1.4, reserve strtof() and strtld().

X3J11 Summary: Reserve strtof and strtld.

X3J11 Response: The Committee has voted for this idea. The functions strtof and strtld cannot be used in a maximally portable program but they are reserved for future use in 4.13.7, **FUTURE LIBRARY DIRECTIONS.**

atof() is problematic from the point of view of naming - it should have been called atod() - and from the point of view of exception handling - it has no universal method to indicate an invalid string argument. It offers nothing that can't be efficiently obtained from strtod(). atof() and atol() should be dropped from the Draft despite their ubiquity, to encourage new code to be written robustly with strtod() and strtol(). Of course, compiler vendors will not need to be told to continue to retain atof() and atol() in their libraries indefinitely.

Recommendation: Footnote atof() and atol() as obsolescent and so list them in 4.13.6.

X3J11 Summary: Deprecate atof and atol.

X3J11 Response: The Committee discussed this proposal but decided against it. There are situations where something simple like atol and atof are useful, and they have existed for a long time.

Comment #20, Section 4.10.2: require two random number generators

The Rationale makes two valid points about random number generators, that repeatability is highly desirable on different implementations, and that different algorithms may be efficient on different machines, but the proposed standard fails to draw the obvious inference: two random number generators should be mandated, one a specified good algorithm that can be implemented to produce identical results with tolerable efficiency on all machines, and another of similar calling sequence that may implement any reliable algorithm of maximal efficiency on a particular machine. Where the first algorithm can be implemented very efficiently, the second name may be only a synonym.

Such duality makes manifest the best way to achieve portability, and the best way to achieve efficiency.

It might be a convenience to provide a standard way to retrieve the current random number generator seed in order to restart a particular sequence later - a grand() corresponding to srand().

Recommendation: Require rand() to return a specific value for a specific argument, by specifying an acceptable algorithm. Define frand() which has the same calling sequences as rand() but implements a random number generator deemed "best" by the implementer; frand() may be the same as rand(). Define srand() and fsrand() to set the current seeds, and grand() and fgrand() to return the current seeds.

RAND_MAX limits rand() but frand() is limited by FRAND_MAX >= RAND_MAX.

I have not followed recent work on portable random number generators and therefore have no recommendation for or against the particular random number algorithm currently in the Draft. When I've needed such a random number generator I have used the one described by L. Schrage in ACM TOMS, 6/79.

X3J11 Summary: Require *two* random number generators.

X3J11 Response: This concerns matters beyond the scope of X3J11. The Committee has discussed this issue and decided that there are too many implementation-specific details surrounding predictable random number generators.

Comment #21, Section 2.2.4.2: <float.h> has too many names, not enough information

The Rationale states that the constants in <float.h> were derived from a similar section of the Fortran 8x proposal. The C committee may not have been aware that those corresponding Fortran proposals are not universally accepted. Tom MacDonald, the principal instigator of <float.h> in C, has indicated that his intent is not to canonize the specific contents of the current Draft's <float.h> but rather to insure that C's floating-point model and inquiries matched Fortran's to the greatest possible extent, to promote easy code conversion. Unfortunately C is ahead of Fortran 8x in the standardization process and is likely to get even further ahead.

Recommendation: Remove the <float.h> specification from the C Draft for now or indicate its tentative status by placing it in the Common Extensions, Future Directions, or Rationale. State a future direction goal of providing a C analog of the final Fortran-8x floating-point characterization.

A proposal for <float.h> and corresponding Fortran follows in Appendix #2.

X3J11 Summary: <float.h> has too many names and not enough information.

X3J11 Response: The Standard must accommodate a variety of architectures. We believe that the <float.h> header provides useful information for a broad class of implementations.

Comment #22, Section 2.2.4.2: long double > minimal float

The committee recognized the need for three integer types, and in order that they be useful, prescribed that implementations providing a minimal short int must also provide a long int of greater precision.

The committee also recognized the usefulness of three floating-point types, and specified that double have more guaranteed precision than float, but made no corresponding prescription on exponent range. Thus all floating-point types may be implemented with the same exponent range, which might not exceed that of VAX or IBM 370 double precision.

Thus an implementation conforming to the current Draft need not even allow an efficient emulation of a common hand-held calculator. This does not facilitate portability. It would be better to prescribe

FLT_DIG	>= 6
DBL_DIG	>= 10
LDBL_DIG	>= DBL_DIG
-FLT_MIN_10_EXP	>= 37
FLT_MAX_10_EXP	>= 37
-DBL_MIN_10_EXP	>= 99
DBL_MAX_10_EXP	>= 99
-LDBL_MIN_10_EXP	>= -DBL_MIN_10_EXP
LDBL_MAX_10_EXP	>= DBL_MAX_10_EXP

As simple as these requirements are - less stringent than IEEE 854's - they exclude D format, the usual VAX double-precision hardware floating point. So from the point of view of practical politics, rather than for any technical reason, some must be relaxed:

-DBL_MIN_10_EXP	>= 37
DBL_MAX_10_EXP	>= 37
-LDBL_MIN_10_EXP	>= 99
LDBL_MAX_10_EXP	>= 99

This relaxed requirement implies that existing VAX code written for "double" can continue to execute with no performance degradation, but that "long double" must be provided on VAXes by other means than D format.

Recommendation: Change the requirements to:

-LDBL_MIN_10_EXP	>= 99
LDBL_MAX_10_EXP	>= 99

X3J11 Summary: Require the **long double** exponent range to exceed the minimal **float** exponent range.

X3J11 Response: The Standard must accommodate a variety of architectures. There are many floating-point implementations that would be non-conforming if these recommendations were adopted. The Committee tries to include as many implementations as possible.

Comment #23, Section 3.1.3.1: compiler conversion same as run-time

In the normal situation in which the compile-time and run-time environments are the same, it is essential that floating-point constant syntax acceptable to the compiler also be acceptable to strtod() and scanf(), and the results of conversion should be identical. This is accomplished automatically if the compiler uses strtod() to convert ascii to binary floating point.

In principle, compilers should accept at least "infinity" and "nan" in situations where numerical floating-point constants are allowed. When the target of compilation does not support infinity, a compilation warning and the largest magnitude should be generated. When the target does not support non-numeric representations such as NaN, Reserved Operand, or Indefinite, a compilation error should be generated.

Recommendation: Wherever a floating-point constant is permitted, a compiler must correctly accept any string acceptable to scanf %lf.

The foregoing recommendation may well be unattractive to compiler implementers who envision difficulty distinguishing a numeric token "infinity" from another kind of identifier "infinitx". About the same benefit would accrue to programmers, however, if casts from strings to floating-point (and for orthogonality, integral) types were permitted. Thus (double) "infinity" will have the same effect as strtod("infinity") except that it will be evaluated at compile time, if possible.

Given the latter feature, the restrictions on compile-time expression evaluation proposed earlier could be relaxed:

a) *double x=3.14;* could always be evaluated at compile time with any side effects disregarded, but

b) *double x=(double)"3.14";* could only be evaluated at compile time if free of side effects.

Alternative Recommendation: Allow casts to floating-point or integral types from explicit strings. The casts may be evaluated at compile time if free of side effects in the run-time environment. Such casts to integral types (type)"string" are equivalent to (type)atol("string") while such casts to floating-point types are equivalent to strtof("string"), strtod("string"), or strtld("string").

X3J11 Summary: Compile-time floating conversion should be the same as at run time.

X3J11 Response: The Committee has voted for this idea. There is a requirement that any compile-time floating-point constant must be converted identically to strtod or scanf. Both these functions refer to the language description of floating-point constants in 3.1.3.1. However there is no requirement to support *NaNs* and infinities.

Comment #24, Section 3.3.2.2 and 3.5.4.3: no implicit declarations

Implicit declarations are a bane of reliable Fortran programming, and C almost uniformly prohibits them, but makes an exception for functions which may be implicitly defined to return integer values, and function parameters which may be implicitly defined to be int. It would make as much sense to treat all undefined variables as implicitly defined ints, on the grounds that reduces the programmer's burden of petty tasks.

The only justification for implicit definitions is prior practice. Prior bugs amply justify this simple change:

Recommendation: Prohibit implicit function type declarations, implicit function parameter type declarations, and invocations of functions with no prototypes or conflicting ones. Because function prototypes must always be in scope at the time of function invocation, remove the rules for implicit conversion of char, short, and float parameters.

The simplification in the Draft is considerable and remarkable. It eliminates knotty questions such as whether the following should be, and are, compatible types:

```
float f();
float g(int);
```

The Draft presumably intends that functions declared to return "float" always return a value rounded to float precision, and returned in the same way; therefore the foregoing declarations are compatible. While "as if" permits returning a float value in a double container, however, that must be done uniformly: in an implementation contemplating backward compatibility, f()'s returned value must not be placed in a double or float container according to whether f() is declared in old or new style, for then the declarations would be incompatible. The Draft or Rationale must speak forcefully to this point so that such implementations are clearly not conforming.

Function prototypes are justifiably regarded by X3J11 as one of its greatest contributions to ANSI C relative to previous common practice. While the desire to provide compatibility with previous practice is laudable in general, in the case of mixed implicit and explicit declarations it leads to questions like the foregoing which have no good answer. Better not standardize a poor one.

Furthermore, older programs which used functions in a consistent way (that passes lint, for instance) can be upgraded largely automatically to include the necessary prototypes. Tools to perform that conversion will probably be widely available. Vendors who perceive a need for even greater backward compatibility will provide such compatibility in a form appropriate to their own previous practice.

X3J11 Summary: Prohibit implicit declarations.

X3J11 Response: This proposal would invalidate too much existing source code. In your example both declarations **float f();** and **float g(int);** are compatible types.

Comment #25, Section 3.5.4.2: fix arrays

I know no C translation that's as clear as the following Fortran code:

```
      SUBROUTINE MATMUL(X,LX,Y,LY,Z,LZ,NX,NY,NZ)
      REAL X(LX,*),Y(LY,*),Z(LZ,*)
      DO 1 I=1,NX
            DO 2 J=1,NZ
            SUM=0
                  DO 3 K=1,NY
3                 SUM=SUM+X(I,K)*Y(K,J)
2           Z(I,J)=SUM
1     CONTINUE
      END
```

Code like this is at the heart of most of the major portable Fortran libraries of mathematical software

developed over the last twenty years. The declared leading dimensions of X, Y, and Z are not known until runtime.

The closest that C can provide is an illusory equivalence by passing, instead of an array X, a dope vector - an array of pointers to X's rows. Creating such a dope vector from an array could be facilitated by mandating a variety of library functions. Equivalent functionality can be obtained in traditional C by treating all arrays as one-dimensional and doing the subscripting "by hand", so that it's harder to get right and harder to optimize, either on the caller's side or the callee's.

The Draft, like traditional C, disallows the equivalent

```
void matmul(x,lx,cx,...)
int lx, cx;
double x[lx][cx] ;
```

unless lx and cx are known at compile time. GNU CC, however, allows variably-dimensioned arrays to be passed as parameters or declared as local variables even more generally; the only requirement seems to be that the array bounds be evaluatable on function entry.

The goal is not to duplicate Fortran's features exactly, but rather to insure that portable linear algebra libraries are as easy to create in C as in Fortran.

The section numberings in the Draft and Rationale are out of synch in the declarations subchapter.

Recommendation: Adopt GNU-CC's treatment of variably-dimensioned arrays, permitting array dimensions of parameters and automatic variables to be determined at run time when needed.

A further-reaching scheme to provide conformant arrays is outlined in Appendix #1 below.

X3J11 Summary: Permit array dimensions of parameters and autos to be determined at run time.

X3J11 Response: The Committee discussed this proposal but decided against it. This invention would have far-reaching implications such as creating pointers to conformant arrays and pointer arithmetic with those pointers. Also, variable argument list processing is more complicated with variably dimensioned arrays.

Comment #26, Section 3.7.1 and A.6.5.9: standardize Fortran-77 interface

Even if many of the other suggestions in this document were adopted, there would still be a variety of situations in which Fortran numerical code is preferable to C:

1) when it's already written;
2) when it uses complex variables;
3) when it uses multi-dimensioned arrays of varying sizes.

I don't advocate adding complex data types to C because they are an easy extension in C++, and arrays are discussed in another comment. Very limited Fortran interfacing is described as a common extension, but much more can be done. Standardizing an interface to Fortran-77 would complete the job of making C the most useful framework for scientific computation.

Recommendation: Add a new environmental section 2.2.5 describing <fortran.h>:

External Fortran-77 FUNCTIONS and SUBROUTINES may be declared in C. <fortran.h> defines certain aspects of a Fortran interface in a portable fashion although the aspects themselves vary among implementations. Certain simple Fortran types are defined in terms of equivalent C types in this fashion:

```
#define INTEGER int
#define LOGICAL unsigned
#define REAL float
#define DOUBLEPRECISION double
#define FUNCTION
#define SUBROUTINE void
```

Note that the definitions for a particular C implementation are chosen to be appropriate for the particular Fortran compiler that C compiler chooses to support. Omitting these definitions indicates that a C compiler supports no Fortran compiler.

Fortran FUNCTION and SUBROUTINE declarations are allowed wherever external C functions may be declared, and consist of the reserved word FORTRAN followed by a function type and function name, followed by the parameter declarations. Parameters must be valid C types although normally the <fortran.h>-defined macros would be used for clarity. Note that the keyword "fortran" entirely in lower case is also defined as equivalent to "FORTRAN" entirely in upper case. The following Fortran-77 subprograms

```
INTEGER FUNCTION IFUNC ( A, B )
INTEGER A
DOUBLEPRECISION B
...

SUBROUTINE LSUB ( C )
LOGICAL C
...
```

correspond to the following declarations in a C program

```
FORTRAN INTEGER FUNCTION IFUNC ( INTEGER A, DOUBLEPRECISION B) ;
FORTRAN SUBROUTINE LSUB ( LOGICAL C) ;
```

and as an example, on some Unix systems those declarations are equivalent to the following conventional C function declarations:

```
extern int IFUNC_( int *A, double *B) ;        extern void LSUB_( unsigned *C) ;
```

but the equivalent C types and function names are implementation-defined.

A C compiler recognizes FORTRAN function invocations in C code and treats them according to the conventions of its supported Fortran compiler. Since all parameters to FORTRAN functions are pointers, an actual parameter that is not a pointer is copied to a temporary and a pointer to the temporary passed instead. Thus an invocation like

```
if (IFUNC(a,&b) == 0) ...
```

would cause the same external reference as a declared external C function, whose name in the Unix example would be IFUNC_. If the parameter a is a pointer to an INTEGER, then it is passed directly; if the parameter a can be cast to an INTEGER, then it is so cast, then copied to a temporary and a pointer passed to the temporary. The type of b, on the other hand, must be DOUBLEPRECISION and its pointer is passed directly. If INTEGER is not equivalent to int, then either the return value of IFUNC, of type INTEGER, or the 0, of type int, will be converted according to the usual rules prior to the comparison.

Note that Fortran-77 CHARACTER types, COMPLEX function values, and EXTERNAL function parameters may have no direct correspondents in C and the parameter and function value conventions may not even be expressible in C. Unix Fortran compilers derived from f77 are often implemented entirely in C but older systems usually implemented Fortran before C.

Finally, if the FORTRAN reserved word is also allowed in declarations of function parameters, then pointers to Fortran functions, whether coded in Fortran or C, can be passed to C functions:

```
g( FORTRAN INTEGER FUNCTION F(...) )
```

Then g will be compiled to produce suitable code for invoking F.

The burden on C compilers not wishing to support Fortran is light:

1 They must provide <fortran.h>, which may be empty.

2 They must respect the FORTRAN reserved word and FORTRAN function declarations which are otherwise legitimate, such as

```
FORTRAN int IFUNC_( int *A, double *B) ;
```

3 They must convert value to reference parameters in FORTRAN function invocations in the manner described above.

X3J11 Summary: Standardize a FORTRAN-77 interface.

X3J11 Response: This concerns matters beyond the scope of X3J11. Attempting this could force us to standardize the interface to many languages.

Comment #27, Section 3.8.8 and 2.2.4.2: predefine generalized precision macros

The generalized precision proposal in Fortran 8x is complicated and controversial, yet responsive to a widely-recognized need among implementers of portable numerical software. The same need is most widely felt in the C community as a question like this: what is the most efficient integral type that contains all integers in the range [-99999,99999] ?

Recommendation: To portably declare such types, add the following predefined macros to 3.8.8:

__int__(p) is evaluated to the name of the smallest signed int type that contains all the signed p-digit integers, i. e. the interval [-(10**p)+1,10**p-1].

__unsigned__(p) is evaluated to the name of the smallest unsigned int type that contains all the p-digit unsigned integers, i. e. the interval [0,10**p-1].

__float__(p,r) is evaluated to the name of the smallest floating-point type that contains all the p-digit signed integers without rounding, i. e. all the integers in the interval [-(10**p)+1,10**p-1], and contains 10**r and 10**-r within its range of positive model numbers.

__INTEGER__(p) is evaluated to the Fortran name of the type corresponding to __int__(p,r), or to INTEGER if no Fortran compiler is supported.

__REAL__(p,r) is evaluated to the Fortran name of the type corresponding to __float__(p,r), or to REAL if no Fortran compiler is supported.

__COMPLEX__(p,r) is evaluated to the Fortran name of the complex type whose components are __float__(p,r), or to COMPLEX if no Fortran compiler is supported.

All the precision- and range-dependent macros fail, terminating compilation, if the precision or range requirements can't be met by any supported type.

Recommendation: In order that the C pre-processor itself may be written portably in C, the C run-time library should provide certain corresponding string-valued functions that return the appropriate type for a particular implementation, or a null pointer on failure. Add the following in <float.h> to implement the corresponding cpp macros:

```
char * int__(int p);
char * unsigned__(int p);
char * float__(int p; int r);
char * INTEGER__(int p);
char * REAL__(int p ; int r);
char * COMPLEX__(int p ; int r);
```

X3J11 Summary: Predefine generalized-precision macros.

X3J11 Response: The Committee discussed this proposal but decided against it. The Committee has the difficult task of deciding what new ideas can be added to C. We felt this idea has merit but could not be included in the Standard.

Comment #28, Section 4.5: specific mathematical library functions

Certain functions in the mathematical library have misguided specifications. The general issue of how to deal with exceptional conditions is treated later: the following comments apply to specific cases in which numerical results and error indications are to be returned.

Non-model arguments: The introduction should explain that the specifications for these functions only apply to model number arguments. There are good examples like MC68881 or 4.3 BSD to follow for extending to IEEE non-model numbers, but other architectures are far too various to specify. Here and later when I suggest an "undefined" result for various situations I mean "defined by the underlying hardware system" where possible, rather than "the C run-time library may implement any capricious response whatever".

X3J11 Summary: Math function specifications should apply only to model number arguments.

X3J11 Response: This concerns matters beyond the scope of X3J11. Specifications for non-model numbers is considered beyond the scope of the Standard. The description applies to model numbers only.

ERANGE: Overflow and underflow are exceptional conditions in the sense that they signal that rounding errors have occurred that are "larger than normal" and may not have been properly accounted for in the design of the program. The error indication should distinguish between overflow and underflow; SVID calls these OVERFLOW and UNDERFLOW. If a numerical result is to be specified at all, it should be the same as that for a multiplication or division producing a similar overflowed or underflowed value, thereby allowing some uniformity in treatment.

X3J11 Summary: Distinguish between overflow and underflow.

X3J11 Response: Quality of implementation is beyond the scope of the Standard. The Standard makes no requirement for distinguishing between overflow and underflow.

EDOM: "Domain errors" include those for which a function of a real variable has no defined value or has a singularity; SVID calls these DOMAIN and SING. The error indication should distinguish between these cases. NaN is an appropriate return value in the former case, signed infinity in the latter; on architectures lacking these, the return value should correspond to 0.0/0.0 and 1.0/0.0.

X3J11 Summary: Distinguish between *NaN* and *singularity*.

X3J11 Response: Quality of implementation is beyond the scope of the Standard. The Standard imposes no requirement on the implementation of *NaNs*.

EIMPL: "Implementation errors" are a way of indicating shortcomings in a particular implementation without confusing them with limitations inherent in a particular floating-point format or intrinsic to a particular mathematical function. They seem to be common in implementations of fmod and trigonometric functions. If the C standard is to sanction such shortcomings at all, they should be distinctively labeled.

X3J11 Summary: Use a distinctive label such as **EIMPL** for "implementation errors".

X3J11 Response: This proposal conflicts with too much prior art. The Committee decided against adding EIMPL because there is no prior art.

fmod arguments: z = fmod(x,y) for positive finite x and y is appropriately, uniquely, and exactly defined as that z of minimum magnitude which has the same sign as x and differs from x by an integral multiple of y. The phrase "integral multiple" is chosen to emphasize that that multiple need not fit in an int, a long, or any other storage format, for it need not be explicitly computed. Implementations that choose to compute fmod(x,y) by a *formula* like

```
x - y * (int)(x/y)
```

should return the EIMPL error indication in cases when the formula does not implement the definition, such as when x/y overflows int format.

The Rationale's wording *the implementation of this function is properly by scaled subtraction rather than division* may shed more darkness than light, being open to easy misinterpretation; the hazardous formula above could be interpreted as a subtraction from x of y scaled by (int)(x/y).

Recommendation: In the Draft, substitute

> z = fmod(x,y), for finite x and finite nonzero y, is uniquely and exactly defined as that z of minimum magnitude which has the same sign as x and differs from x by an integral multiple of y.

X3J11 Summary: Fix the definition of fmod.

X3J11 Response: The Committee has voted for this idea. The Standard already defines fmod the way you desire.

Recommendation: In the Rationale, substitute a statement that naive implementations of the formula x - y * (int)(x/y) do not conform; the phrase "integral multiple" emphasizes that the multiple need not fit in an int, a long, or any other storage format, for it need not be explicitly computed. Instead, fmod can be computed in principle by subtracting ldexp(y,n) from x, for appropriately chosen decreasing n until the remainder is between 0 and x.

fmod(x,y), like (x % y), is almost always invoked for a specific constant y like 10, 24, or $\pi/2$, and never for y==0 except in error. Reasonable-sounding defenses can be given for both fmod(x,0) == x and fmod(x,0) == 0 ; the paradox arises because of the division by zero lurking in the background. Therefore no prudent programmer would rely on the numerical result of fmod(x,0), so it's best left undefined (to allow for a NaN result when available). If an error indication is desired, the appropriate one is that for 0.0/0.0 - a domain error.

Trigonometric arguments: The proper treatment of large trigonometric arguments has long been a source of confusion. One satisfactory approach is to always perform a "correctly rounded" argument reduction by generating a variable-precision approximation to π that is large enough to reduce a particular argument correctly (4.3 BSD VAX version); another is to reduce all arguments with an approximation to π that is fixed to a precision greater than or equal to the precision of the largest supported floating-point format (4.3 BSD IEEE version; MC68881).

An unsatisfactory approach is to compute a reduced argument as fmod(x,π/2) using a defective fmod of the type mentioned above. Such algorithms have a catastrophic internal boundary at which (int) x/(π/2) overflows int format; SVID describes such results as "total" or "partial" loss of significance but since the real issue is a particular implementation, EIMPL as described above is a more appropriate response. The satisfactory algorithms mentioned in the previous paragraph don't contain any such internal boundary; the fixed-precision π algorithm gradually loses accuracy for increasing arguments, but that loss is practically undetectable, except by comparison with a variable-precision π algorithm, because all essential identities and relationships continue to hold for large arguments as well as small.

The Draft's *a large magnitude argument may yield a result with little or no significance* not only overlooks that such significance loss may occur with rather small arguments such as sin(n * π), but encompasses algorithms with catastrophic failures at an internal boundary equally with satisfactory

fixed-precision algorithms. If exceptional cases are to be detected by the math library functions then the former failures should be so detected, but not the latter. SVID mandates that the former be detected but misclassifies the occurrence as EDOM rather than EIMPL.

Recommendation: Remove the sentences about results with little or no significance. Add a statement in the Rationale:

> Trigonometric argument reduction should be performed by a method that causes no catastrophic discontinuities in the error of the computed result. In particular methods based solely on naive application of a formula like
>
> $$r = x - (2 * \pi) * (\text{int})(x/(2 * \pi))$$
>
> are not conforming.

X3J11 Summary: Fix the range reduction specification.

X3J11 Response: The Committee believes this is clear enough as is. 4.5.1 states that the behavior of these functions is defined for all representable values of *their* input arguments. An implementation that truncates to **int** is not Standard conforming.

Inverse trigonometric argument ranges: The inverse trigonometric functions are mathematically multiple-valued and therefore a particular branch needs to be specified for computation. However the proposed standard inadvertently goes too far and in effect specifies tight bounds and directions on rounding errors at the endpoints of the ranges of acos, asin, atan, and atan2. In contrast, nothing in the standard requires that 1+1 be close to 2!

Appropriate wording for the standard is to refer to the *approximate ranges* [0,P] or [–P/2,+P/2] where P is a machine-representable number whose value is close to π but not necessarily the closest machine-representable number. Thus the intended branch is made apparent without setting any unrealistic expectation that the end points will be correctly rounded to π or $\pi/2$. The approximate range for atan2 is [–P,+P].

Recommendation: In the Rationale, explain that the intervals in question are approximate and intended to indicate the proper branch without setting an expectation that the functions will be rounded any more accurately at the endpoints of their ranges than elsewhere.

X3J11 Summary: Clarify ranges of inverse trig functions.

X3J11 Response: Quality of implementation is beyond the scope of the Standard. The inverse trigonometric functions can be implemented this way.

atan2(0.0,0.0): atan2 is generally used in conversion from rectangular coordinates to the polar coordinate argument θ. The θ associated with the origin is of little consequence, and computing it seldom indicates an error. Conformal mapping applications can treat edges symmetrically if atan2($\pm$0.0,$\pm$0.0) is allowed to take any of the values ± 0 or $\pm\pi$. Such methods aren't available on non-IEEE machines, so the best choice for the C standard is to restrict the numerical result of atan2(0.0,0.0) to the approximate range $[-\pi,+\pi]$ but otherwise undefined. On IEEE machines, setting atan2(0,0) to NaN would be a mistake in that it would have the consequence that a NaN rather than zero would be the result of multiplying any normal complex number in polar coordinates (ρ,θ) by the origin in polar coordinates (0.0, NaN).

X3J11 Summary: Improve atan2(0,0) specification.

X3J11 Response: This was accepted as an editorial change to the Standard. Your point is exactly correct; the January 1988 draft did not correctly reflect the intent of the Committee. It was specifically desired to allow implementations the freedom to return a value such as 0 or *NaN* instead of a domain error for **atan2(0,0)**. It is not clear that representing the ambiguous result as *NaN* would necessarily be a "mistake"; atan2 can be used for purposes other than conversion to polar coordinates.

What about hypot? Atan2(y,x) and hypot(x,y)=sqrt(x*x+y*y) are complementary components of rectangular to polar coordinate conversion and should either both be standardized or neither.

Recommendation: Standardize hypot():

4.5.6.5 The hypot function

#include <math.h>
double hypot (double x, double y);

Description

The hypot function computes sqrt(x*x+y*y), avoiding however gratuitous intermediate underflow and overflow.

X3J11 Summary: Standardize hypot.

X3J11 Response: This was considered to be an invention of limited utility. Although this function has uses, the Committee felt because this function did not exist in the base document and because of the limited scope in which it is useful that there was insufficient reason to add it to the Standard.

log(0.0) and log10(0.0): The proper interpretation of these values is that of a singularity. No overflow or underflow occurs. Since for real variables exp(−∞) == +0, it follows that log(+0) == −∞, which can be standardized as -HUGE_VAL. The Rationale's statement that a *range* error occurs for IEEE 854 compatibility is unaccountable, since IEEE 854 doesn't specify elementary transcendental functions like log(); the somewhat related IEEE 854 appendix function logb(0.0) is specified as a singularity (domain error) like -1.0/0.0. The MC68881 flogn instruction, for instance, follows the intent of the IEEE committees by returning −∞ and signaling singularity (division by zero, a domain error). I believe that this case was intended to meet IEEE expectations (domain rather than range error) but was overlooked by accident or misunderstanding at the December 1987 ANSI meeting.

Recommendation: Change the error to domain.

X3J11 Summary: Make log(0) a *domain error.*

X3J11 Response: The Committee discussed this proposal but decided against it.

pow(0.0,0.0): There are good arguments that pow(x,0.0) should be 1.0 for all x, without any exceptional indication; 1.0 is the value most useful, most often. For instance, (x+y)**n computed as a sum of terms involving (x**n-j)(y**j), 0 <= j <= n ; this formula readily works as intended if x**0 is 1 regardless of x. Such arguments are not universally accepted, so if the ANSI C committee is unprepared to accept them, it would be appropriate for the standard not to specify pow(0.0,0.0). At the December 1987 meeting, the Draft was changed to specify a domain error if the result was not representable in an implementation.

X3J11 Summary: Improve pow(0,...) specification.

X3J11 Response: The Committee has voted for this idea. An implementation is at liberty to return 1.0 for pow(0,0).

pow(0.0,negative integral value): 0.0**-1.0 is just 1.0/0.0, and if an exception is to be specified it should be like that of 1.0/0.0.

"integral values" in floor() and ceil(): Ambiguous language in Fortran-77's definition of aint() and anint(), much like the Draft C standard's definition of ceil() and floor(), has misled some implementers into thinking that such functions could be correctly implemented by converting the argument to a long or int and then converting back to double.

Recommendation: Expand the definition of ceil:

The ceil function returns the smallest integral value in double format greater than or equal to x, even though that integral value might not fit in a variable of type int or long. Thus ceil(x) == x for all x sufficiently large in magnitude; for IEEE 754 double precision, that includes all x such that fabs(x) >= 2**52.

X3J11 Summary: Expand the definition of ceil.

X3J11 Response: The Committee believes this is clear enough as is. 4.5.1 states that the behavior of these functions is defined for all representable values of their input arguments. An implementation that truncates to **int** is not Standard conforming.

frexp and modf: The Rationale mentions that these functions almost went the way of ecvt et al. Too bad they didn't since frexp and modf are not essential for a mathematical library implementation and modf in particular is a source of confusion because it has been specified in a variety of ways in the past. The standard's definition of frexp is only appropriate for binary floating point. In any case a more useful function is int ilogb(double d), defined at least for model numbers so that ilogb(J*b**(e+1-p)) = e; in other words, ilogb is the exponent of its argument adjusted so that ilogb(1.0) = 0. For flexibility, ilogb(0.0) can be allowed to return any value less than ilogb of the smallest positive representable number.

Recommendation: Delete frexp and modf from the Draft.

X3J11 Summary: Delete frexp and modf.

X3J11 Response: The Committee discussed this proposal but decided against it. frexp has found uses in important portability applications; it need not be thought of as a "mathematical" function.

ldexp might be more useful if its name were more clearly related to its function; *scalen()* indicates its common use and its relation to the *scalb()* function defined in the IEEE 754 appendix. The standard's definition of ldexp, like frexp, is only appropriate for binary floating point. Is this an accident or does it reflect an intent to favor binary floating point? C has always favored binary integer arithmetic but has not explicitly inhibited non-binary floating point; the definitions proposed for <float.h> envision arbitrary floating-point bases.

Recommendation: Change the definition of ldexp to

ldexp(double x, int e) computes x*(DBL_RADIX**e) , avoiding explicit exponentiation or multiplication.

X3J11 Summary: Change the definition of ldexp.

X3J11 Response: The Committee discussed this proposal but decided against it.

The current Draft's ldexp() and frexp() are of no use or interest whatever on machines with decimal floating point; scaling by powers of two has no special significance in that case. Floating-point scaling by ldexp corresponds most to integer shifting; and is easier to implement in floating-point hardware than, for instance, multiplication; it's an atomic MC68881 operation. So the cleanest approach is

Recommendation: Remove 4.5.4.3 and other references to ldexp(). Extend 3.3.7 to permit the left-hand expression of << and >> to have floating-point type, with the following semantics: the result of ((double) x) << e is x*(DBL_RADIX**e), and the result of ((double) x) >> e is x*(DBL_RADIX**(-e)), with corresponding definitions for float and long double, preferably computed efficiently without any explicit exponentiation or multiplication. Unlike the case with integer shifts, however, the result is defined for any e, although it may overflow or underflow; the result in those cases is defined by the underlying system, just as the result of a multiplication is.

X3J11 Summary: Delete ldexp in favor of allowing a floating-point value to be shifted.

X3J11 Response: The Committee discussed this proposal but decided against it. ldexp has found uses in important portability applications; it need not be thought of as a "mathematical" function.

matherr(): X3J11's conclusions about the SVID function matherr() are right on the mark; too bad some of the criteria mentioned in the Rationale weren't applied uniformly to other aspects of C:

* The style of error handling is redundant on IEEE systems.
* matherr()-style error handling gets in the way of porting Fortran code to C.
* User-written matherr() might require re-entrant libm.
* A mechanism like signal() would be more flexible (but would still imply re-entrant libm).

A common initial reaction to matherr() messages on standard error is "how do I get rid of these?" *Even so the matherr() mechanism is better than the errno mechanism that ANSI C retained!*

matherr() may have been derived from the best practice for 1970's-vintage mainframes, represented by Cody's error handling for his FUNPACK transcendental function package. The IEEE 854 standard presumably indicates Cody's current thinking on this subject.

Comment #29, Section 4.13.4: standard functions predefine generic operators

When Fortran was being invented, both division and square root were commonly implemented as subroutines - sometimes rather similar ones. However the first Fortran character set, apparently a subset of a commercial typewriter's, had a / character available but not a mathematical square root symbol. So Fortran evolved using / for division and sqrt() for square root, and in this respect has been followed by other algebraic languages. But division and square root are both operators, and rather similar in some respects - the use of operator or functional notation being a historical accident. Other cases are similar, such as unary – and fabs(). In many modern hardware implementations sqrt is just an atomic operation like division; likewise IEEE arithmetic was designed so that both unary – and fabs() may be accomplished by one-bit operations. On an MC68881, all the following C "functions" could be implemented in one or two instructions by disregarding errno exception reporting requirements: fabs, fmod, ldexp, sqrt, exp, log, log10, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh.

The point relevant to C is that many of its standard numerical functions should really be thought of as operators (designated, like *sizeof*, with words rather than special typographical symbols) rather than external functions (although some might be implemented with invisible external functions). Of course, that means that a program can't pass a pointer directly to fabs or sqrt, but nobody laments that it is tedious to create a function that performs division in order to create and pass a pointer to it. Similarly there is no great loss in requiring anyone wanting to create a pointer to a function that computes sqrt to go to the trouble of creating his own function that calls the predefined sqrt, then passing a pointer to that function.

Unlike fully reserved identifiers, operators designated by predefined names should be undefined by the compiler upon encountering a programmer-supplied declaration for that name. The compiler should issue a warning however at that point since such redeclarations are not generally commendable programming practice. In the context of ANSI C, a reasonable alternative would be to prohibit such redeclarations that don't specify the parameter type list as quite likely accidental, while silently allowing redeclarations that specify the parameter type list since then accidents are far less likely.

A consequence of generic treatment is that there is no need to reserve additional function names fabsf, fabsl, sqrtf, sqrtl, etc., as contemplated in section 4.13. It also means that, being generic functions known to the compiler, constant expressions involving them could be evaluated at compile time. An additional consequence of generic treatment is that these functions could take an optional extra argument which is useful in providing better exception handling, as described later. Finally, providing uniform names for numerical operators reduces the effort required to convert numerical code from one precision to another. Imagine the cries of distress if the / operator were required to be written /f for float operands and /l for long double.

Generalized precision: the proposed Fortran 8x standard includes a new facility which allows a variable's floating-point precision and exponent range to be declared in a machine-independent way at compile time, thereby solving a common problem in portable mathematical software: how can I declare variables of a particular precision? For instance, if the equivalent of 10 decimal digits and 2 decimal exponent digits are required to simulate a hand-held calculator, there is no direct way to insure that such variables are declared in the shortest format (float, double, long double) with the necessary precision and range. Instead experiments must be performed to determine the appropriate type, then the source code changed (at best, perhaps only a few macro definitions must be changed). I hope that Fortran 8x facility will prove successful in use so that a comparable one may be incorporated in a future C standard. Providing such a facility is greatly simplified if the language's standard numerical operators and functions are provided by generic operators that derive their types from the types of their operands.

What about built-in functions (Section 4.1.6 footnote)? The standard requires implementations to provide actual functions for these operators on demand but also allows implementations to define them by default as macros and, ultimately, to expand them in-line. Thus an implementation that exploited that allowance would be able to obtain some of the advantages of the generic treatment outlined above. But the main advantage of in-line treatment is lost since such built-in functions must meet the same errno-based exception handling requirements as the corresponding actual functions.

Algebraic max and min and absolute-magnitude mmax and mmin would also make excellent candidates for definition as generic operators. Fortran experience provides ample justification.

Recommendation: Define all the libm functions, as well as square, max, min, mmax, and mmin as generic operators applicable to integral and floating-point types; except that the transcendental functions acos, asin, atan, atan2, cos, sin, tan, cosh, sinh, tanh, exp, log, and log10 apply only to floating-point types.

The following less comprehensive alternative restricts operators to those which can be meaningfully applied to integral as well as floating-point types, and for which correctly-rounded results can be economically obtained, thus excluding transcendental functions and base conversion.

Alternate Recommendation: Define at least abs, max, min, mmax, mmin, ceil, floor, sqrt, hypot, mod, ldexp if << is not extended to floating, and square if a power operator is not defined, as generic operators in functional notation operating upon any numerical values.

X3J11 Summary: Define math and additional functions as generic operators.

X3J11 Response: This would run counter to the historical "spirit of C". The Committee discussed this and decided to keep these as explicit functions.

Comment #30, Section 4.5.1: make numerical exception handling uniform

The traditional C scheme for exception handling has little to recommend it: there is no guarantee that numerical exceptions are even detectable for the most common floating-point operators, yet the standard would enforce the unreliable global errno mechanism for the comparatively less frequently used elementary transcendental functions. Such an approach promotes tradition rather than safety or efficiency.

The problems associated with errno are mentioned in the Rationale but not universally appreciated, especially by those who think of portability in the context of complete source-code applications rather than object libraries of functions destined to be combined with other such libraries from other sources.

A provider of a library of software functions for a general C environment will usually have to get along without the benefit of errno since otherwise that library could not run reliably in an application which exploits asynchronous signal handlers. Those signal handlers may call functions in some other library from a different provider which set and reset errno according to their own needs. Since in general the errno implementation may be no more than a single global variable, in the face of asynchronous signal handlers it can't be relied on to contain the value it had in the previous statement. The discipline that allows errno to be used reliably would have to be applied consistently in an application built of components from diverse sources - not too likely. The same synchronization problems arise when multiple processors share memory.

What about matherr()? SVID requires matherr(), a user-definable function, to be called in every situation in which errno would be set to EDOM or ERANGE. Matherr() doesn't provide any more flexibility than errno but concentrates error-handling code in one place. Matherr's fatal shortcoming is that only one such global function can exist per application; thus object libraries intended for general application can't rely on matherr without excluding other libraries that might want to exploit their own versions of matherr().

A language standard contemplating implementation only on modern systems equipped with IEEE arithmetic might require that all floating-point exceptions relating to finite representation (inexact, underflow, overflow), singularity (division by zero), and domain violations would be detected and treated uniformly whether they arose in atomic hardware operations or software subroutines. Such architectures permit efficient exception detection and treatment. Although IEEE floating point status

flags are often implemented in a global variable that suffers some of the limitations of global errno, the IEEE standard permits other implementation methods and, more importantly, returns appropriate numerical values in addition to exception status.

Values like infinity and NaN are not available on many systems, so errors must be communicated by other means than the returned value. Old architectures can't simultaneously accommodate both efficiency and safety, however, and a language standard that attempts to accommodate those architectures in both those aspects must pay the price in duplication of facilities. I discern three alternative methods for providing such facilities:

Dual types
Dual operators
Specify efficiency, defer safety

METHOD #1: Dual types. In addition to the native floating-point types, require all implementations to also provide floating-point types, operations, and functions corresponding to the IEEE 854 standard for floating-point arithmetic independent of word length or radix. Thus it would be possible to write fairly portable safe code on all ANSI C implementations. Even more portable would be to require IEEE 754 binary single- and double-precision implementations. In either case such IEEE arithmetic would have to be implemented in software on systems in which it is not supported by hardware.

METHOD #2: Dual operators. Instead of requiring dual floating-point types, require generic floating-point operators to accept an optional additional argument which is essentially a pointer to a local errno. These dual operators allow error status to be associated with specific operations. The exception handling requirements would be quite strictly specified for invocations which specify the error-reporting pointer, and for efficiency, completely undefined if that pointer is missing. For generic operators denoted functionally, the syntactic extension would be modest:

```
double sqrt(x,perr)
double x;
int *perr;
```

*perr is the local errno for this operation.

For the other operators the additional syntax would be more striking:

```
double _div(x,y,perr)
double x, y;
int *perr;
```

suggests an operator with a functional notation that returns the double quotient x/y with a local error indication in *perr.

This proposal encompasses far less than the previous; it allows the same data types to be operated upon by efficient means most of the time and more carefully when necessary.

METHOD #3: Specify efficiency, defer safety: C++ allows overloading operators and might therefore be a more appropriate vehicle than C for the safety-oriented extensions described above. Then the C standard should anticipate such extensions by not defining exception handling at all for the mathematical functions, consistent with its treatment of the common operators. This is my preferred method for reasons discussed in the next section.

Recommendation: Remove all references to EDOM, ERANGE, errno, and exceptional return values from the descriptions of the 4.5 functions and the 4.10.1.4 function strtod().

X3J11 Summary: Remove exception specifications from the descriptions of the math functions.

X3J11 Response: This proposal would invalidate too much existing source code. Although you are right in recognizing that errno is inadequate, there is too much existing code that would break.

What should languages attempt to standardize about floating-point operators and functions?

Most of what follows has been argued better elsewhere. See, for instance, Kahan and Coonen's "The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments," contained in the compendium edited by J. K. Reid, *The Relationship between Numerical Computation and Programming Languages*, North-Holland 1982.

Some of the mathematical functions traditionally included with C are overspecified with respect to exceptional conditions, in the sense that results have been specified for operations like fmod(x,0.0) that are unlikely to arise except in programs that are primarily intended to test compliance with the specification, such as the SVID validation suite. A consequence of such an approach is that efficiency in the normal case is lost by checking for unlikely exceptions for the purpose of providing unreliable exception handling, such as a global errno. This slight payoff has a disproportionate cost; a modern high-performance hardware fmod implementation may lose half its performance in checking for exceptional arguments and setting errno. That modern implementation will be safer, too, by returning a NaN instead of a numerical result.

Most designers of portable software desire that systems return reasonable results and continue when exceptional conditions occur that the designers anticipated; the desirable response for unanticipated exceptions might be termination with debugging output. Thus my recommendation above that exception responses be unspecified is painful but appropriate given the improbability that a C standard would specify default exception handling for ordinary operators like * or /.

The Draft C standard follows other language standards in making no requirement at all on the common floating-point operators for any operands, and making no requirement for functions evaluated in their non-exceptional domain. One reason for this is that it's easy to specify and test behavior at isolated exceptional points, and difficult to specify and test normal behavior. But it's the "normal" behavior which is most refractory to deal with effectively in a portable fashion. Language standardizers are usually reluctant to specify anything outside of the control of the compiler and its run-time library, despite that imperfections in the hardware or operating system adversely affect the language user's ability to specify his computation reliably and economically.

Thus the Draft C standard specifies something about sqrt(–1.0) - EDOM - but nothing about sqrt(1.0). An appropriate expectation for sqrt(–1.0) and other exceptional functions is that they *should not* return an unexceptional result and proceed as if nothing had happened. What they *should* do varies among systems: a NaN and a retrospective diagnostic are appropriate on IEEE implementations, a Reserved Operand on a VAX, a diagnostic message on some other systems, possibly abnormal process termination in closed systems. Any universally-prescribed response is necessarily constrained to the capability of the least common denominator of all these systems and so is a poor choice for many.

Unlike a software vendor, a person who programs for his own use usually uses different languages at different times, but typically on only one machine. That user tends to think of the floating-point and elementary functions, provided in hardware or coded in assembler by the hardware manufacturer, as part of the underlying conceptual machine rather than characteristic of a language implementation, and he'd prefer that those aspects be the same in all languages he programs in. So from that point of view, the standard C mathematical library specification would do well to follow the precept expressed in the Rationale, "many operations are defined to be how the target machine's hardware does it." Thus when I suggest an "undefined" result in various situations I mean "defined by the underlying system" where possible, rather than "the C run-time library may implement any capricious response whatever". Of course in many Unix systems the C run-time library and the Unix kernel below it *are* the underlying system, but that situation is hardly universal.

Otherwise, what's an appropriate specification for sqrt(1.0)? Here are some possibilities:

1) sqrt(x) is correctly rounded
   for all finite representable x >= 0
2) sqrt(x) has error < 1 unit in the last place
   for all finite representable x >= 0
3) sqrt(x) has error < 1 unit in the last place
   for all model numbers x >= 0
4) sqrt(x) increases monotonically
   for x >= 0
5) sqrt is correct for perfect squares, i.e.
   sqrt(x*x) == x for all x = i where 0 <= i <=
   (int) sqrt(DBL_RADIX**DBL_MANT_DIG);

Properties 4) and 5) are the most useful, although they are implied by 1), the most stringent specification. Mathematical software providers could provide long lists of desirable properties for each floating-point operator and library function.

But some compiler suppliers would object that they can't or won't meet any specific accuracy requirement. Furthermore it is fairly difficult to verify compliance to any of these specifications, and more difficult for transcendental functions than for rational functions like division or algebraic ones like sqrt.

The IEEE floating-point committees, faced with considerations like these, soon discovered that a *descriptive* standard for floating-point arithmetic, that somehow allowed every existing irregular floating-point implementation to claim conformance, would have no useful properties. Instead, in order to promote future compatibility and portability, it was necessary to develop a *prescriptive* standard, not necessarily corresponding to any existing implementation, that provides significant usable compatibility among new implementations.

A related issue is whether standardization properly comes before common use or vice-versa. Prior to IEEE arithmetic, no floating-point standard existed or was likely to exist that was common to more than one manufacturer and its clones. Common usage was only possible after the standard was developed. Similarly providers of portable C software are loathe to exploit any idiomatic C feature in a particular compiler not canonized by some standard. Consequently providers of C compilers have no individual incentive to provide such features until standardized. Finally, X3J11 is reluctant to standardize features without a demonstration of "prior art". To exit this tendency to regress toward the mean, somebody must lead.

The ANSI C Rationale acknowledges many difficult issues that arose from conflict between the descriptive and prescriptive approaches to standardization, and like many language standards the result often tends toward the descriptive. Since the ANSI C committee is unlikely to want to transcend previous language standards by closely specifying the entire C numerical environment, it should avoid prejudicing the issue in any of the ways discussed previously. Then another body may properly build upon the ANSI C standard by adding to it, rather than subtracting from it, in order to specify a suitable environment for scientific computation.

Acknowledgements

Although they could not agree with all the foregoing recommendations, the following persons generously contributed detailed criticism of earlier versions of these comments:

Nelson Beebe	Beebe@SCIENCE.UTAH.EDU
David Gay	research!dmg
John Gilmore	hoptoad!gnu
Earl Killian	mips!earl
Tom MacDonald	Cray Research
Doug McIlroy	research!doug
Richard O'Keefe	quintus!ok
Scott Turner	sdti!turner
Jim Valerio	omepd!radix!jimv
David Wolverton	houxs!daw

Appendix #1: A Proposal for Conformant Arrays

Simply allowing variably-dimensioned arrays as in GNU CC would be just as {un}satisfactory as Fortran's facilities. Conformant arrays are a greater change to the language but possibly a smaller change to existing implementations. Richard O'Keefe has kindly provided the following outline of a C array facility providing better error immunity than Fortran has traditionally provided:

Conformant arrays in C are much as in ISO Pascal or in Turing, with parameter declarations like

```
void matmul(double a[p][q], double b[q][r], double c[p][r])
   {
      int i, j, k;
      double t;
      for (i = p; --i >= 0; )
         for (j = r; --j >= 0; ) {
            t = 0.0;
            for (k = q; --k >= 0; ) t *= a[i][j]*b[j][k];
            c[i][j] = t;
         }
   }
```

If p, q, and r are #defined to be constant expressions, this is already legal, so we need one more thing to indicate that p,q,r are being defined here, not used. Consider the following:

```
declarator:    ...
        |    declarator '[' subscript_spec ']'
        ;

subscript_spec: 'auto' identifier
              | constant_expression
              | /* empty */
              ;
```

where /* empty */ is only allowed as the first subscript_spec, and auto id is only allowed in a function header. To avoid having to specify what happens if auto x appears several times but the arguments don't agree with that, make it illegal, so the first suggestion would have to be written

```
void matmul(double a[auto ar][auto ac], /* ar >= p, ac >= q */
        double b[auto br][auto bc], /* br >= q, bc >= r */
        double c[auto cr][auto cc], /* cr >= p, cc >= r */
        int p,          /* 0 <= p <= min(ar,cr) */
        int q,          /* 0 <= q <= min(ac,br) */
        int r)          /* 0 <= r <= min(bc,cc) */
  {
     int i, j, k;
     double t;
     for (i = p; --i >= 0; )
        for (j = r; --j >= 0; ) {
           t = 0.0;
           for (k = q; --k >= 0; ) t *= a[i][j]*b[j][k];
           c[i][j] = t;
        }
  }
```

A conformant array may be passed as a parameter to a function provided the function's prototype was visible to confirm that a conformant array parameter was intended.

The simplest way of treating sizeof is to rule it out: if the description of 'a' involves any auto s, you can't apply sizeof to it. So given a parameter

```
float fred[auto f1][auto f2][10];
```

sizeof fred and sizeof fred[1] would be illegal, but sizeof fred[1][2] and sizeof fred[1][2][3] would be legal.

X3J11 Summary: Support *conformant arrays*.

X3J11 Response: The Committee discussed this proposal but decided against it. This invention would have far-reaching implications such as creating pointers to conformant arrays and pointer arithmetic with those pointers. Also, variable argument list processing is more complicated with variably dimensioned arrays.

Appendix #2: A Proposal for <float.h>

The discussion that follows indicates the direction that Fortran and C should follow to align their floating-point inquiry and manipulation functions in useful fashion. First, the restriction on six character external identifiers in C must be removed; a holdover from the earliest Fortran systems, it can't survive the advent of Fortran 8x, as the identifiers in the following tables demonstrate.

X3J11 Summary: Remove the 6-character external identifier restriction, which is doomed even for FORTRAN.

X3J11 Response: The Committee has reaffirmed this decision on more than one occasion. This has been discussed many times in the past and there is no hope for this limit to be changed, because that would invalidate some implementations.

X3J11 Summary: Improve <float.h>.

X3J11 Response: The Committee discussed this proposal but decided against it. We believe that <float.h> is beneficial for many floating-point applications.

The proposal that follows is partly built upon a model for floating-point numbers that includes a normalized subset of the machine-representable numbers in most implementations. The model numbers consist of 0 and the set of rational numbers

```
J * b**(e+1-p)
```

where J is an integer satisfying

```
b**(p-1) <= abs(J) <= (b**p)-1
```

and e is an integer satisfying

```
emin <= e <= emax
```

Thus the model numbers exclude –0, subnormal numbers, signed infinity, and NaNs on IEEE systems. The model number parameters b, p, emin, and emax are used similarly in the IEEE 854 standard, although 854 puts no lower bound on abs(J), in order that the formula encompass subnormal numbers and zero.

Model numbers represent the least common denominator subset of all popular floating-point implementations. W. S. Brown (ACM TOMS 12/81) envisioned further limiting p, emin, and emax to exclude representable numbers whose arithmetic participation is unsatisfactory, such as partially underflowed results in CDC-6600 descendants and partially overflowed results on Crays.

The following proposes a set of numerical operators in functional notation. The notation is C but the application to Fortran is obvious. Corresponding Fortran 8x draft operators are listed if they exist. Numerical operators, rather than functions, are chosen specifically to encourage compile-time evaluation for constant operands. Numerical operators, rather than C #defines, are chosen to facilitate generalized precision inquiries in which the outcome can't be determined at compile time, which is the way Fortran 8x's generalized precision sometimes works. If that facility were modified so that it were fully determinate at compile time, then the environmental inquiries could become #defines as envisioned in the current X3J11 draft's <float.h>.

The first set describes environmental inquiries corresponding to C's sizeof; the operands are actually types, represented by variables declared to be of those types. The type "fp" means any of float, double, or long double.

Proposed Floating-Point Environmental Inquiry Operators		
Proposed Operator	**Corresponding X3J3/S8.104**	**Description**
		Model Number Parameters
int RADIX(fp x)	RADIX	Base b
int SIGNIFICANCE(fp x)	DIGITS	Precision p
int MIN_EXPONENT(fp x)	MINEXPONENT	Minimum exponent emin
int MAX_EXPONENT(fp x)	MAXEXPONENT	Maximum exponent emax
fp HUGE_MODEL(fp x)	HUGE	Largest positive model number b**(emax+1) * (1 – b**–p)
fp HUGE_REPRESENTABLE(fp x)		Largest positive representable number; +∞ on IEEE systems
int EXPONENT_RANGE(fp x)	EFFECTIVE_EXPONENT_RANGE	Largest power P such that 10**P and 10**–P are in the range of positive model numbers
int EFFECTIVE_PRECISION(fp x)	EFFECTIVE_PRECISION	Largest decimal precision no more precise than type fp
int INPUT_PRECISION(fp x)		Largest decimal precision unchanged by input|output
int OUTPUT_PRECISION(fp x)		Smallest decimal precision unchanged by output|input
fp PERFORMANCE(fp x)		Relative performance of floating-point multiplication
int IEEE_754()		1 if conformance claimed to ANSI/IEEE Std 754-1985
int IEEE_854()		1 if conformance claimed to ANSI/IEEE Std 854-1987

The following operators allow manipulation of floating-point quantities and thus correspond to traditional C's ldexp() and frexp().

Proposed Floating-Point Manipulation Operators		
Proposed Operator	**Corresponding X3J3/S8.104**	**Description**
int EXPONENT(fp x)	EXPONENT	Exponent of model number e+1–p
fp SIGNIFICAND(fp x)	FRACTION	Significand J*b**(–p)
int ISNAN(fp x)		1 if x is NaN, else 0
fp NEXT_MODEL(fp x,y)	NEAREST	Next model number from x in direction y
fp NEXT_REPRESENTABLE(fp x,y)		Next representable number from x in direction y
fp SCALE(fp x, int n)	SCALE	Scaling by power of base replacing ldexp(x,n)

The following functions (not operators) are required in the C run-time library only to support corresponding generalized precision macros in the C preprocessor.

Proposed Generalized Precision Functions for C	
Proposed Function	**Corresponding 3.8.8 macro**
char * int__(int p)	__int__(p)
char * unsigned__(int p)	__unsigned__(p)
char * float__(int p; int r)	__float__(p,r)
char * INTEGER__(int p)	__INTEGER__(p)
char * REAL__(int p ; int r)	__REAL__(p,r)
char * COMPLEX__(int p ; int r)	__COMPLEX__(p,r)

The following superfluous operators should be deleted.

Operators Proposed for Deletion		
Operator X3J3/S8.104	**Corresponding X3J11/88-002**	**Description and Reason for Deletion**
SETEXPONENT		Insert exponent - easy to perform with SCALE
TINY	FLT_MIN	Smallest model number - easy to find from NEXT_MODEL(0.0,1.0)
EPSILON	FLT_EPSILON	b**(1–p) - easy to compute from formula or NEXT_MODEL(1.0,2.0)–1.0
SPACING		absolute spacing between model numbers - NEXT_MODEL(x,HUGE_MODEL)–x
RRSPACING		reciprocal of relative spacing - roughly x/(NEXT_MODEL(x,HUGE_MODEL)–x)
	FLT_ROUNDS	Rounding method - too difficult to characterize usefully

PERFORMANCE() is needed in a generalized-precision implementation that supports arbitrary precision to indicate when a program has "fallen off the end" of hardware-supported precision into much slower software-supported precision. PERFORMANCE(x) is just the ratio of the performance of floating-point multiplication for x's type to the performance for Fortran REAL type (float for C's that don't support any Fortran compiler).

IEEE_754, IEEE_854: These operators allow code to portably determine if IEEE arithmetic is implemented, in order to exploit its properties when available, and proceed more cautiously otherwise. They are operators (rather than #defines) only for compatibility with the other operators in this section.

ISNAN() is needed to allow distinguishing floating-point variables containing numerical values from those that don't. On systems with NaNs, some or all NaNs usually generate exceptions when touched by floating-point arithmetic and so may be robustly filtered otherwise only by machine-dependent bit field manipulations.

NEXT_REPRESENTABLE() and *NEXT_MODEL()* are more generally applicable than constants such as EPSILON. Such constants need be determined only once at the start of a computation, so any performance difference between a constant and a function evaluation is not significant. The IEEE 754 interpretation of nextafter(x,y) - the nearest {representable,model} number from x in the direction y - is more convenient than Fortran 8x's, which deduces the direction from the sign of y rather than the relationship of y to x.

INPUT_PRECISION and *OUTPUT_PRECISION* are claims about an implementation's internal floating-point format and about the conversion sequences (scanf|printf) and (printf|scanf) for model numbers. A decimal string of no more than INPUT_PRECISION significant figures, that may be read by scanf to produce a model number without overflow or underflow, may then be printf'ed with

INPUT_PRECISION significant figures to produce a string of unchanged numerical value, although the format may differ. A string of no more than INPUT_PRECISION significant figures will be reproduced if printed in the same format, unaltered by the limited internal precision of the variable in which the converted value was stored.

A model number printed with at least OUTPUT_PRECISION significant figures will be reproduced unchanged by scanf. Thus printing a numerical value with OUTPUT_PRECISION significant figures specifies the internal value unambiguously.

The mathematical theory underlying conversion between binary and decimal was summarized in Sterbenz's *Floating-Point Computation*, Prentice-Hall, 1974. For IEEE 754 single-precision arithmetic, INPUT_PRECISION is 6, OUTPUT_PRECISION is 9, while for double precision, INPUT_PRECISION is 15, and OUTPUT_PRECISION is 17.

Fragments like the following might be used in a test of an implementation's INPUT_PRECISION and OUTPUT_PRECISION:

```
#include <float.h>

double
randommodel()
{
    /*
     * computes a "random" model number chosen from the whole space of
     * model numbers
     */
}

testmax()
{
    double      d1, d2;
    char        s1[40], s2[40];
    do {
        d1 = randommodel();
        sprintf(s1, "%*.*e", OUTPUT_PRECISION(d1) + 10, OUTPUT_PRECISION(d1) - 1, d1);
        sscanf(s1, "%lf", &d2);
    }
    while (d1 == d2);
    printf(" OUTPUT_PRECISION too low! ");
}

testmin()
{
    double      d1, d2;
    char        s1[40], s2[40];
    do {
        d1 = randommodel();
        sprintf(s1, "%*.*e", INPUT_PRECISION(d1) + 10, INPUT_PRECISION(d1) - 1, d1);
        sscanf(s1, "%lf", &d2);
        sprintf(s2, "%*.*e", INPUT_PRECISION(d1) + 10, INPUT_PRECISION(d1) - 1, d2);
    }
    while (strcmp(s1, s2) == 0);
    printf(" INPUT_PRECISION too high! ");
}
```

How should <float.h> constants be specified, and how many?

Intellectual economy argues that the number of these constants should be minimized by reliance on nextrepresentable and nextmodel. But unless these functions are categorically specified for every variety of machine, portable software may be reluctant to rely upon them in lieu of constants. The constants must be very carefully specified - algorithmically perhaps - with a quite likely result that the algorithm will be subject to misinterpretation. For instance, the X3J11 Draft specifies FLT_EPSILON as the minimum float x > 0 such that (1.0 + x) != 1.0, namely pow(b,1–p). Given the freedom of expression evaluation in C, very much smaller float x than intended will satisfy (1.0 + x) != 1.0 if that expression is evaluated in double as C's expression evaluation rules allow. Perhaps what's meant is ((float)(1.0 + x)) != 1.0. Even that is not pow(b,1–p) when rounding resolves ambiguous cases to even. Perhaps what's really meant is that FLT_EPSILON is just pow(b,1–p). But then how do you usefully apply that quantity? What started out as a laudable attempt at portability has ended up being a source of subtle bugs instead. Thus the EPSILON, SPACING, and RRSPACING of Fortran 8x seem to be more trouble than they're worth.

Instead of EPSILON, an environmental inquiry of equivalent utility to mathematical software experts but probably more comprehensible to novices would be *fp MAX_INTEGRAL(fp x)* which returns the maximum integral value M such that all integral values whose magnitudes do not exceed M are representable exactly in the format of x. This is usually b**p.

FLT_ROUNDS: The variation of floating-point rounding algorithms on non-IEEE machines is too great to encapsulate in a simple definition. Better to require applications to fabricate the specific test that matters to them, such as

```
double eps, oneminus, nextrepresentable();

eps = 1.0 – nextrepresentable( 1.0, 0.0) ;
oneminus = (double)(1.0 – 0.5 * eps) ;
if (oneminus == 1.0)
      { looks like some kind of rounding }
else
      { looks like some kind of chopping }
```

This code depends on the cast in "oneminus = (double)..." and fails if the indicated rounding does not occur as might be the case in some traditionally-minded implementations that might allocate oneminus to a long double register. FLT_ROUNDS might be worth specifying in a form like the following:

> A value of {FLT,DBL,LDBL}_ROUNDS in the following table is a claim about an implementation's arithmetic for the conventional floating-point operators: for representable (not just model) x and y, whenever computed (x ±*/ y) or x ±*/= y raises no exception except possibly IEEE inexact, the computed result is always rounded correctly to a representable value:
>
> | FLT_ROUNDS value | Corresponding rounding method |
> |---|---|
> | 0 | Rounded toward nearest, halfway case rounded to even. (IEEE default) |
> | 1 | Rounded toward zero. |
> | 2 | Rounded toward negative infinity. |
> | 3 | Rounded toward positive infinity. |
> | 4 | Rounded toward nearest, halfway case rounded away from zero. (VAX) |
>
> If {FLT,DBL,LDBL}_ROUNDS is undefined or defined to another value, then no claim about the implementation's rounding is to be inferred.

Note that separate statements for each precision make sense in environments where one precision is implemented in fast sloppy hardware and another in slow careful software. No point requiring the

software to be as sloppy as the hardware. Note further that including division in the list of correctly-rounded operators excludes Cray, while including addition and subtraction excludes IBM 370, which *almost* always rounds toward zero.

Instead of a numerical {FLT,DBL,LDBL}_ROUNDS, it might be more useful to have one or more string-valued macros {FLT,DBL,LDBL}_TYPE that defined the type of arithmetic claimed from a list of the most common floating-point implementation styles:

IBM 370 single	IBM 370 double	IBM 370 extended
	Cray single	Cray double
VAX F	VAX D	
	VAX G	VAX H
IEEE 754 single	IEEE 754 double	IEEE 754 double-extended
IEEE 854 single	IEEE 854 double	IEEE 854 double-extended
Other		

Then portability would really be aided - among these common types - by allowing a program to confidently know the run-time arithmetic rather than trying to infer it indirectly by ostensibly machine-independent definitions.

Appendix #3: Why does traditional C treat float the way it does?

Although it's well known that floating-point type conversions on a PDP-11 were inconvenient, the traditional C float treatment was principally intended to provide robust floating-point programming, with an intended side effect of reducing the number of math library functions required: C was designed before generic intrinsic Fortran functions were standardized.

As to robustness, especially when large arrays have to be stored in 32-bit "float" for storage economy, it's desirable to evaluate expressions involving floats in higher precision; that's what the MC68881 and similar processors were designed to do, with better error bounds and no loss in performance. But other systems have different properties; if "float" were 32-bit hardware, and "double" were 64-bit software, the performance penalty for evaluating in double precision would be enormous compared to the benefit. Much signal processing and graphics computation derives no benefit whatever from double precision computation. In modern high-performance hardware, the difference in cost between 32 and 64-bit floating-point division, square root, and elementary functions is still at least a factor of two. So traditional C evaluation is rather costly to mandate generally.

Therefore the Draft is correct in not specifying default expression evaluation too closely, so that expressions can be evaluated in the narrowest format allowed by the usual conversions OR in any wider format, according to a particular implementation's properties. The Draft is also correct in specifying that explicit and implicit casts always cause a rounding to the target type if narrower than the type in which the expression to be cast was evaluated.

Even if float expressions were all evaluated in float, there's an unexpected cost in traditional C to transmitting float parameters and function results. ANSI C allows floats to be passed directly if the parameter list is specified, but traditional C requires shameful kluges in order to implement, for instance, an efficient Fortran compiler and library.

Possibly ANSI C should permit flexibility by defining operators which force other styles of evaluation for their operand expressions:

* *minimal ANSI C = traditional Fortran:* operation precision is that of more-precise operand
* *traditional C:* all float operands and operations in double
* *widest needed:* operation precision is that of most precise operand in entire expression; see papers by Corbett in SIGPLAN 17#12 or farnum@renoir.berkeley.edu, "Compiler Support for Floating-Point Computation."
* *widest available:* all operands and operations in long double

In every case, explicit or implicit cast should cause rounding to that type to occur.

X3J11 Summary: Why does traditional C treat floating-point the way it does?
X3J11 Response: A specific proposal is needed before action can be taken.