

# **The New C Standard** (Excerpted material)

---

**An Economic and Cultural Commentary**

**Derek M. Jones**

derek@knosof.co.uk

## 6.4.4.2 Floating constants

floating constant  
syntax

```

floating-constant:
    decimal-floating-constant
    hexadecimal-floating-constant
decimal-floating-constant:
    fractional-constant exponent-partopt floating-suffixopt
    digit-sequence exponent-part floating-suffixopt
hexadecimal-floating-constant:
    hexadecimal-prefix hexadecimal-fractional-constant
    binary-exponent-part floating-suffixopt
    hexadecimal-prefix hexadecimal-digit-sequence
    binary-exponent-part floating-suffixopt
fractional-constant:
    digit-sequenceopt . digit-sequence
    digit-sequence .
exponent-part:
    e signopt digit-sequence
    E signopt digit-sequence
sign: one of
    + -
digit-sequence:
    digit
    digit-sequence digit
hexadecimal-fractional-constant:
    hexadecimal-digit-sequenceopt .
    hexadecimal-digit-sequence
    hexadecimal-digit-sequence .
binary-exponent-part:
    p signopt digit-sequence
    P signopt digit-sequence
hexadecimal-digit-sequence:
    hexadecimal-digit
    hexadecimal-digit-sequence hexadecimal-digit
floating-suffix: one of
    f l F L

```

### Commentary

The majority of *floating-decimal-constants* do not have an exact binary representation. For instance, if `FLOAT_RADIX` is 2 then only 4% of constants having two digits after the decimal point can be represented exactly (i.e., those ending .00, .25, .50, and .75).

Unlike an *integer-suffix*, a *floating-suffix* specifies the actual type, not the lowest rank of a set of types (not that floating-point types have rank).

Hexadecimal floating constants were introduced to remove the problems associated with translators incorrectly mapping character sequences denoting decimal floating constants to the internal representation of floating numbers used at execution time. The potential mapping problems only apply to the significand, so a decimal representation can still be used for the exponent (requiring a hexadecimal representation for the exponent would have made it harder for human readers to quickly gauge the magnitude of a constant and created a lexical ambiguity, e.g., would the character sequence `p0x1f` be interpreted as ending in the *floating-suffix* `f` or not).

The exponent is always required for the hexadecimal notation, unlike decimal floating constants, otherwise, the translator would not be able to resolve the ambiguity that occurs when a `f`, or `F`, appears as the last

character of a preprocessing token. For instance, `0x1.f` could mean `1.0f` (the `f` interpreted as a suffix indicating the type `float`) or `1.9375` (the `f` being interpreted as part of the significand value).

The *hexadecimal-floating-constant* `0x1.FFFFFFFp128f` does not represent the IEC 60559 single-format NaN. It overflows to an infinity in the single format.

## C90

Support for *hexadecimal-floating-constant* is new in C99. The terminal *decimal-floating-constant* is new in C99 and its right-hand side appeared on the right of *floating-constant* in the C90 Standard.

## C++

The C++ syntax is identical to that given in the C90 Standard.

Support for *hexadecimal-floating-constant* is not available in C++.

## Other Languages

Support for *hexadecimal-floating-constant* is unique to C. Fortran 90 supports the use of a *KIND* specifier as part of the floating constant. Fortran also supports the use of the letter *D*, rather than *E*, in the exponent part to indicate that the constant has type **double** (rather than `real`, the single-precision default type). Java supports the optional suffixes *f* (type **float**, the default) and *d* (type **double**)

## Coding Guidelines

Mapping to and from a hexadecimal floating constant, and its value as a floating-point literal, requires knowledge of the underlying representation. The purpose of supporting the hexadecimal floating constant notation is to allow developers to remove uncertainty over the accuracy of the mapping, of values expressed in decimal, performed by translators. Developers are unlikely to want to express floating constants in hexadecimal notation for any other reason and the guideline recommendation dealing with use of representation information is not applicable.

?? representation information using

Dev ??

Floating constant may be expressed using the hexadecimal floating-point notation.

The advantage of hexadecimal floating constants is that they guarantee an exact (when `FLT_RADIX` is a power of two) floating value in the program image, provided the constant has the same or less precision than the type.

For the same rationale as integer constants, there is good reason why most floating constants should not appear in the visible source.

?? integer constant not in visible source

Cg 842.1

No floating constant, other than `0.0` and `1.0`, shall appear in the visible source code other than as the sole preprocessing token in the body of a macro definition.

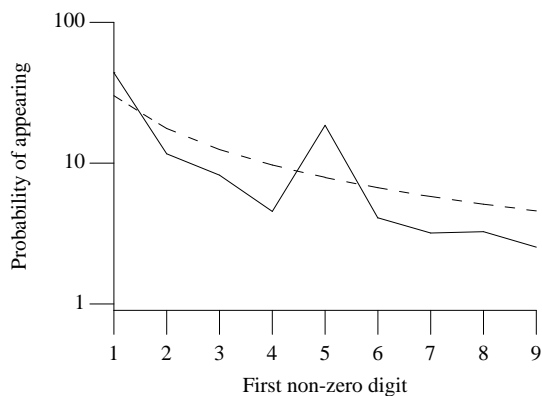
## Usage

Exponent usage information is given elsewhere. Also see elsewhere for a discussion of Benford's law and the first non-zero digit of constants ( $\chi^2 = 1,680$  is a very poor fit).

exponent integer constant usage

**Table 842.1:** Occurrence of various *floating-suffixes* (as a percentage of all such constants). Based on the visible form of the `.c` and `.h` files.

Suffix Character Sequence	.c files	.h files
none	98.3963	99.7554
F/f	1.4033	0.1896
L/l	0.2005	0.0550



**Figure 842.1:** Probability of a *decimal-floating-constant* (i.e., not hexadecimal) starting with a particular digit. Based on the visible form of the .c files. Dotted line is the probability predicted by Benford's, i.e.,  $\log(1 + d^{-1})$ , where  $d$  is the numeric value of the digit.

**Table 842.2:** Common token pairs involving *floating-constants*. Based on the visible form of the .c files.

Token Sequence	% Occurrence of First Token	% Occurrence of Second Token	Token Sequence	% Occurrence of First Token	% Occurrence of Second Token
, <i>floating-constant</i>	0.0	20.4	<i>floating-constant</i> /	5.8	1.8
= <i>floating-constant</i>	0.1	15.7	*= <i>floating-constant</i>	6.3	1.6
* <i>floating-constant</i>	0.2	12.5	<i>floating-constant</i> *	6.8	0.1
( <i>floating-constant</i>	0.0	8.8	<i>floating-constant</i> ;	26.5	0.1
+ <i>floating-constant</i>	0.4	7.7	<i>floating-constant</i> )	25.9	0.1
-v <i>floating-constant</i>	0.3	6.7	<i>floating-constant</i> ,	25.8	0.1
/ <i>floating-constant</i>	2.0	6.4			

## Description

significand part A floating constant has a *significand part* that may be followed by an *exponent part* and a suffix that specifies its type. 843

### Commentary

This defines the terms *significand part* and *exponent part*.

whole-number part  
fraction part The components of the significand part may include a digit sequence representing the whole-number part, followed by a period (.), followed by a digit sequence representing the fraction part. 844

### Commentary

A restatement of information given in the Syntax clause. The character denoting the period, which may appear when floating-point values are converted to strings, is locale dependent. However, the period character that appears in C source is not locale dependent.

A leading zero does not indicate an octal floating-point value.

### C++

2.13.3p1 *The integer part, the optional decimal point and the optional fraction part form the significand part of the floating literal.*

The use of the term *significand* may be a typo. This term does not appear in the C++ Standard and it is only used in this context in one paragraph.

## Other Languages

This form of notation is common to all languages that support floating constants, although in some languages the period (decimal point) in a floating constant is not optional.

## Coding Guidelines

The term *whole-number* is sometimes used by developers. A more commonly used term is *integer part* (the term used by the C++ Standard). The commonly used term for the period character in a floating constant is *decimal point*.

A common mathematical convention is to have a single nonzero digit preceding the period. This is a useful convention when reading source code since it enables a quick estimate of the magnitude of the value to be made. There are also circumstances where more than one digit before the period, or leading zeros before and after the period, can improve readability when the floating constant is one of many entries in a table. In this case the relative position of the first non zero digit may provide a useful guide to the relative value of a series of constants, which may be more important information than their magnitudes.

Your author knows of no research showing that any method of displaying floating constants minimizes the cognitive effort, or the error rate, in comprehending them. However, there does appear to be an advantage in having consistency of visual form between constants close to each other in the source. Comprehending the relationship between the various initializers appears to require less effort for `g_1` and `g_2` than it does for `g_3`.

```
1  double g_1[] = {
2      1.2,
3      0.12,
4      0.012,
5      0.0012,
6      0.00012,
7      };
8  double g_2[] = {
9      1.2e-0,
10     1.2e-1,
11     1.2e-2,
12     1.2e-3,
13     1.2e-4,
14     };
15 double g_3[] = {
16     1.2e-0,
17     0.12,
18     1.2e-2,
19     0.0012,
20     1.2e-4,
21     };
```

floating constant  
digit layout

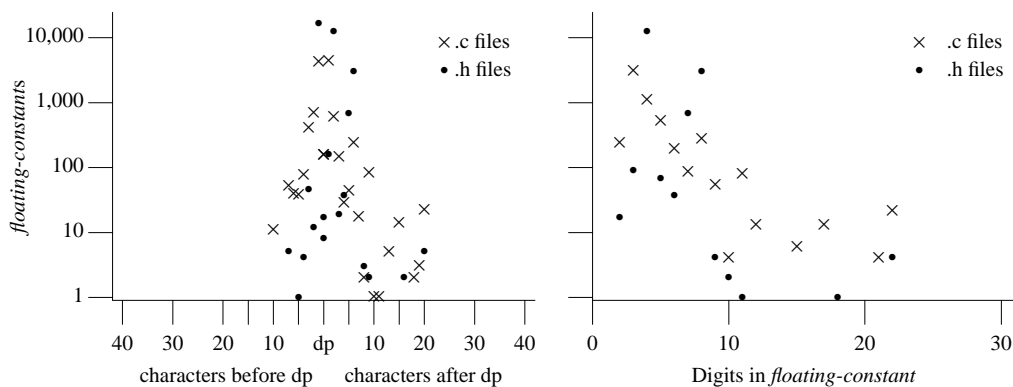
Trailing zeros in the fractional part of a floating constant may not affect its value (unless a translator has poor character to binary conversion), but they do contain information. Trailing zeros can be interpreted as a statement of accuracy; for instance, the measurement 7.60 inches is more accurate than 7.6 inches.

floating constant  
assumed  
accuracy

Leading zeros are sometimes used for padding and have no alternative interpretation. Adding trailing zeros to a fractional part for padding purposes is misleading. They could be interpreted as giving a floating constant a degree of accuracy that it does not possess. While such usage does not affect the behavior of a program, it can affect how developers interpret the accuracy of the results.

Rev 844.1

Floating constants shall not contain trailing zeros in their fractional part unless these zeros accurately represent the known value of the quantity being represented.



**Figure 844.1:** Number of *floating-constants*, that do not contain an exponent part, containing a given number of digit sequences before and after the decimal point (dp), and the total number of digit in a *floating-constant*. Based on the visible form of the .c and .h files.

The components of the exponent part are an *e*, *E*, *p*, or *P* followed by an exponent consisting of an optionally signed digit sequence. 845

### Commentary

A restatement of information given in the Syntax clause.

### C90

Support for *p* and *P* is new in C99.

### C++

Like C90, the C++ Standard does not support the use of *p*, or *P*.

### Other Languages

The use of the notation *e* or *E* is common to most languages that support the same form of floating constants. Fortran also supports the use of the letter *D*, rather than *E*, to indicate the exponent. In this case the constant has type **double** (there is no type **long double**).

### Coding Guidelines

Amongst a string of digits, the letter *E* can easily be mistaken for the digit 8. There is no such problem with the letter *e*, which also adds a distinguishing feature to the visual appearance of a floating constant (a change in the height of the characters denoting the constant). However, there is no evidence to suggest that this choice of exponent letter is sufficiently important to warrant a guideline recommendation. At the time of this writing there is little experience available for how developers view the exponent *p* and *P*. While the prefix indicates that a hexadecimal constant is being denoted, a lowercase *p* offers an easily distinguished feature that its uppercase equivalent does not.

### Example

```

1  double glob[] = {
2      67E9,
3      9e76,
4      };

```

## Commentary

A restatement of information given in the Syntax clause.

## Coding Guidelines

When only one of these parts is present, the period character might easily be overlooked, especially when floating constants occur adjacent to other punctuation tokens such as a comma. This problem can be overcome by ensuring that a digit (zero can always be used) appears on either side of the period. However, such usage is not, itself, free of problems. The period can be interpreted as a comma (if the source is being quickly scanned), causing the digits on either side of the period to be treated as two separate constants. The issue of white space between tokens is discussed elsewhere. In the case of digits after the decimal point, there is also the issue of assumed accuracy of the floating constant.

words  
white space  
between  
844 floating  
constant  
assumed accuracy

## Example

```

1  extern int g(double, int, ...);
2
3  double glob[] = {
4      12.3, .456, 789.,
5      12.3, .456,789.,
6      98.7, 0.654, 321.0,
7      98.7,0.654,321.0,
8      };
9
10 void f(void)
11 {
12     g(1.2, 45, 6., 0);
13     g(1.2, 45, 6,0);
14     g(7.8,90, 1.,2);
15     g(3.4,56, 7.0,8);
16 }
```

---

847 for decimal floating constants, either the period or the exponent part has to be present.

## Commentary

A restatement of information given in the Syntax clause. Without one of these parts the constant would be interpreted as an integer constant.

## Coding Guidelines

Is there a benefit, to readers, of including a period in the visible representation of a floating constant when an exponent part is present? Including a period further differentiates the appearance of a floating constant from that of other types of constants. Developers with a background in numerate subjects will have frequently encountered values that contain decimal points. The exponent notation used in C source code is rarely encountered outside of source code, powers of ten usually being written as just that (e.g.,  $10^2$ ). Because of these relative practice levels, developers are much more likely to be able to automatically recognize a floating value with a constant that contains a period than one that only contains an exponent (which is likely to require conscious attention). However, given existing usage (see Figure 844.1) a guideline recommendation does not appear worthwhile.

automatiza-  
tion

Developers reading source often only need an approximate estimate of the value of floating constants. The first few digits and the power of ten (sometimes referred to as the *order of magnitude* or simply the *magnitude*) contain sufficient value information. The magnitude can be calculated by knowing the number of nonzero digits before the decimal point and the value of the exponent. There are many ways in which these two quantities can be varied and yet always denote the same value. Is there a way of denoting a floating constant such that its visible appearance minimizes the cognitive effort needed to obtain an estimate of its value? The possible ways of varying the visible appearance of a floating constant including:

- Not using an exponent; the magnitude is obtained by counting the number of digits in the whole-number part.
- Having a fixed number of digits in the whole-number part, usually one; the magnitude is obtained by looking at the value of the exponent.
- Some combination of digits in the whole-number part and the exponent.

There are a number of factors that suggest developers' effort will be minimized when small numbers are written using only a few digits before the decimal point rather than using an exponent, including the following:

- Numbers occur frequently in everyday life and people are practiced at processing the range of values they commonly encounter. The prices of many items in shops in the UK and USA tend to have only a few digits after the decimal point, while in countries such as Japan and Italy they tend to have more digits (because of the relative value of their currency).
- *Subitizing* is the name given to the ability most people have of instantly knowing the number of items in a small set (containing up to five items, although some people can only manage three) without explicitly counting them.

subitizing

Your author does not know of any algorithm that optimizes the format (i.e., how many digits should appear before a decimal point or selecting whether to use an exponent or not) in which floating-point constants appear, such that reader effort in extracting a value from them is minimized.

### Example

Your author is not aware of any studies investigating the effect that the characteristics of human information processing (e.g., the Stroop effect) have on the probability of the value of a constant being misinterpreted.

stroop effect

```

1  double d[] = {
2      123.567,
3      01.1,
4      3.333e2,
5      1.23456e8,
6      1111.3,
7      122.12e2,
8      };

```

## Semantics

---

The significand part is interpreted as a (decimal or hexadecimal) rational number;

848

### Commentary

One form is based on the human, base 10, representation of values, the other on the computer, base 2, representation.

### C90

Support for hexadecimal significands is new in C99.

### C++

The C++ Standard does not support hexadecimal significands, which are new in C99.

### Other Languages

While support for the hexadecimal representation of floating constants may not be defined in other language standards, some implementations of these languages (e.g., Fortran) support it.

**Example**

What is the IEC 60559 single-precision representation of 12.345? For the digits before the decimal point we have:

$$12_{10} = 1100_2 \quad (848.1)$$

For the digits after the decimal point we have:

$$\begin{array}{r} .345 \\ \times 2 \\ 0 .690 \\ \times 2 \\ 1 .380 \\ \times 2 \\ 0 .760 \\ \times 2 \\ 1 .520 \\ \times 2 \\ 1 .040 \\ \times 2 \\ 0 .080 \\ \times 2 \\ 0 .160 \\ \dots \end{array}$$

$$.345_{10} = .010001011000010100011111_2 \quad (848.2)$$

Writing the number in normalized form, we get:

$$1100.01011000010100011111 \times 2^0 = 1.10001011000010100011111 \times 2^3 \quad (848.3)$$

Representing the number in single-precision, the exponent bias is 127, giving an exponent of  $127 + 3 = 130_{10} = 10000010_2$ . The final bit pattern is (where | indicates the division of the 32-bit representation into sign bit, exponent, and significand):

$$0 \mid 10000010 \mid 10001011000010100011111 \quad (848.4)$$

What is the decimal representation of the hexadecimal floating-point constant, assuming an IEC 60559 representation of 0x0.12345p0? For the significand we have:

$$.12345_{16} = .00010010001101000101_2 = 1.0010001101000101 \times 2^{-4} \quad (848.5)$$

For the exponent we have:

$$127 - 4 = 123_{10} = 1111011_2 \quad (848.6)$$

which gives a bit pattern of:

$$0 \mid 1111011 \mid 00100011010001010000000 \quad (848.7)$$

Converting this to decimal, the exponent is  $1111011_2 = 123 - 127 = -4_{10}$ ; and restoring the implicit 1 in the significand, we get the value:

$$1.0010001101000101_2 \times 2^{-4} \quad (848.8)$$

$$0.00010010001101000101_2 \quad (848.9)$$

$$(1/16 + 1/128 + 1/2048 + 1/4096 + 1/16384 + 1/262144 + 1/1048576)_{10} \quad (848.10)$$

$$(74565/1048576)_{10} \quad (848.11)$$

$$7.111072540283203125 \dots_{10} \times 10^{-2} \quad (848.12)$$

Taking into account the accuracy of the representation, we get the value  $7.111073_{10} \times 10^{-2}$ .

---

the digit sequence in the exponent part is interpreted as a decimal integer.

849

### Commentary

Even if the significand is in hexadecimal notation, the exponent is still interpreted as a decimal quantity.

**C++**

2.13.3p1 *... , an optionally signed integer exponent, ...*

There is no requirement that this integer exponent be interpreted as a decimal integer. Although there is wording specifying that both the integer and fraction parts are in base 10, there is no such wording for the exponent part. It would be surprising if the C++ Standard were to interpret `1.2e011` as representing  $1.2 \times 10^9$ ; therefore this issue is not specified as a difference.

---

For decimal floating constants, the exponent indicates the power of 10 by which the significand part is to be scaled.

850

### Commentary

This specification is consistent with that for the significand.

---

For hexadecimal floating constants, the exponent indicates the power of 2 by which the significand part is to be scaled.

851

### Commentary

Scaling the significand of a hexadecimal floating constant by a power of 2 means that no accuracy is lost when all the powers of 2 specified by the exponent are representable using the value of `FLT_RADIX`. (This is always true when `FLT_RADIX` has a value of 2, as specified by IEC 60559.) This scaling is performed during translation; it is not an execution-time issue.

**C90**

Support for hexadecimal floating constants is new in C99.

**C++**

The C++ Standard does not support hexadecimal floating constants.

852 For decimal floating constants, and also for hexadecimal floating constants when `FLT_RADIX` is not a power of 2, the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner.

floating constant  
representable  
value chosen

### Commentary

This behavior is different from that specified for integer-to-float conversion. Having introduced hexadecimal floating constants, the Committee could not then restrict their use to implementations having a `FLT_RADIX` that was a power of 2. The advantage of exact representation does not always occur in implementations where `FLT_RADIX` is not a power of 2, but a translator is still required to support this form of floating constant.

int to float  
nearest repre-  
sentable value

When converting a value to a different base the best approximation can be obtained using a finite automaton if the two bases are both an integral power of a common integral root.<sup>[2]</sup> For hexadecimal floating constants this requires that the other base be a power of 2. Unless two bases have this property it is not possible to always find the best approximation, when converting a value between them, using a finite automaton (there are some numbers that require arbitrary precision arithmetic to obtain the best approximation). Thus, it is not possible to find the best  $n$ -bit binary approximation of a decimal real number using a finite automaton.<sup>[2]</sup> For this reason the C Standard does not require the best approximation and will accept the nearest representable values either side of the best approximation (see Figure 852.1).

859 hexadecimal  
constant  
not represented  
exactly

Floating constants whose values have the same mathematical value, but are denoted by different character sequences (e.g., `1.57` and `15.7e-1`), may be mapped to different representable values by a translator. (Depending on how its mapping algorithm works, the standard permits three different possibilities.) In fact the standard does not even require that implementations map the same sequence of characters to the same value internally (although an implementation that exhibited such behavior would probably be considered to have low quality). Neither is there any requirement, in this case, that if the constant value is exactly representable the exact representation be used.

DECI-  
MAL DIG  
conversion  
recommended  
practice

### C90

Support for hexadecimal floating constants is new in C99.

### C++

*If the scaled value is in the range of representable values for its type, the result is the scaled value if representable, else the larger or smaller representable value nearest the scaled value, chosen in an implementation-defined manner.*

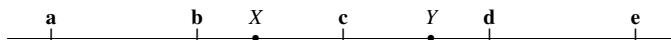
2.13.3p1

### Other Languages

This issue is not C specific. All languages have to face the problem that only a finite number of floating-point values can be represented and the mapping from a string to a binary representation may not be exact.

### Common Implementations

A few implementations continue to use the *student* solution of multiply and add, a character at a time, moving left-to-right through the significand of the floating constant, followed by a loop that multiplies, or divides, the result by 10; the iteration count being given by the value of the exponent. Your author knows of no translator that maps identical floating constant tokens into different internal representations. Mapping different floating constant tokens, which have the same mathematical value, to different internal representations is not unknown.



**Figure 852.1:** The nearest representable value to  $X$  is  $b$ , however, its value may also be rounded to  $a$  or  $c$ . In the case of  $Y$ , while  $d$  is the nearest representable value the result may be rounded to  $c$  or  $e$ .

### Coding Guidelines

There are a number of factors that can introduce errors into the conversion of floating-point tokens (a sequence of characters) into the internal representation used for floating values (a pattern of bits) by the translator, including:

- Poor choice of conversion algorithm by the translator vendor, leading to larger-than-permitted, by the standard, error
- Poor choice of rounding direction when performing conversions
- The finite set of numbers that can be represented in any model of floating-point numbers

Translator errors in conversion of floating constants are most noticeable, by developers, when they have an exact representation in an integer type. In:

```

1  #include <stdio.h>
2
3  void f(void)
4  {
5  double d = 6.0;
6
7  if ((int)d != 6)
8      printf("Oh dear\n");
9  /* ... */
10 }
```

The standard does not require that the character sequence 6.0 be converted to the floating value 6.0. The value 5.9999999 (continuing for a suitable number of 9s) is the smaller representable value immediately adjacent to the nearest representable value—a legitimate result—which, when cast to **int**, yields a value of 5. A translator that chooses to round-to-even, or not round toward zero, would not exhibit this problem.

These coding guidelines deal with developer-written code. Developers do not need to be told, in a guideline, to use high-quality implementation. If developers are in the position of having to use an implementation that does not correctly convert floating constant tokens, then all that can be suggested is that hexadecimal floating constants be used. The error introduced by the model used by an implementation to represent floating numbers has to be lived with. The issue of converting floating values to integer types is discussed elsewhere.

### Example

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5  if ((1.57 != 1.570) || (1.57 != 0.157E1) ||
6      (0.0 != 0.000) || (0.0 != 0.0E10))
7      printf("You can never enter the same stream twice\n");
8  }
```

For hexadecimal floating constants when **FLT\_RADIX** is a power of 2, the result is correctly rounded.

853

### Commentary

If **FLT\_RADIX** is a power of 2, the hexadecimal value has a unique mapping to the representation of the significand (any additional digits, in excess of those required by the type of the floating constant, are used to form the correctly rounded result).

floating ??  
constant  
converted exactly

correctly  
rounded  
result

## Common Implementations

All known implementations have a `FLT_RADIX` that is a power of 2.

854 An unsuffixed floating constant has type **double**.

### Commentary

A floating constant has type **double**, independent of its value. This situation differs from the integer types where the value of the literal (as well as its form) is used to determine its type. There is no suffix to explicitly denote the type **double**.

### Other Languages

Most languages do not have more than one floating-point type. A literal could only have a single type in these cases. A floating constant in Fortran has type **real** by default.

### Common Implementations

Some translators<sup>[1]</sup> provide an option that allows this default behavior to be changed (perhaps to improve performance by using **float** rather than **double**).

### Coding Guidelines

If an unsuffixed floating constant is immediately converted to another floating type, would it have been better to use a suffix? An immediate conversion of a floating constant to another floating type suggests that the developer is being sloppy and is omitting the appropriate suffix. In the case of a conversion to type **long double**, there is also the possibility of a loss of precision. The floating constant token is converted to **double** first, potentially losing any additional accuracy present in the original token, before being converted to **long double**; however, the issues are not that simple.

The precision to which a floating constant is represented internally by a translator need not be related to its type. It can be represented to less or greater precision depending on the value of the `FLT_EVAL_METHOD` macro. A suffix changes the type of a floating constant, but need not change the precision of its internal representation. On the other hand, a cast operation is required to remove any additional precision in a value, but does not have the information needed to provide additional precision.

Explicitly casting floating constants ensures the result is consistent (assuming the cast occurs during program execution, not at translation time, which can only happen if the `FENV_ACCESS` pragma is in the `ON` state) with casts applied to nonconstant operands having floating type.

The disadvantage of specifying a suffix on a floating constant, because of the context in which the constant is used, is that the applicable type may change. The issues involved with implicit conversion versus explicit conversion are discussed elsewhere. An explicit cast, using a typedef name rather than a suffix, is more flexible in this regard.

The following guideline recommendations mirror those given for suffixed integer constants, except that it is specified as a review item, not a coding guideline. The process of resolving whether a suffix or cast is best in light of possible settings for the `FLT_EVAL_METHOD` needs human attention.

Rev 854.1

A floating constant containing a suffix shall not be immediately converted to another type.

Dev 854.1

The use of a macro defined in a system header may be immediately converted to another type.

Dev 854.1

The use of a macro defined in a developer-written system header may be immediately converted to another type, independent of how the macro is implemented.

Dev 854.1

The body of a macro may convert, to a floating type, one of the parameters of that macro definition.

## Example

`FLT_EVAL_METH`

implicit conversion

?? integer constant with suffix, not immediately converted

```

1  #include <stdio.h>
2
3  void f(void)
4  {
5  if ((1.0f / 3.0f) != ((float)1.0 / (float)3.0))
6     printf("No surprises here (float)\n");
7  if ((1.0 / 3.0) != ((double)1.0 / (double)3.0))
8     printf("No surprises here (double)\n");
9  if ((1.0L / 3.0L) != ((long double)1.0 / (long double)3.0))
10     printf("No surprises here (long double)\n");
11 }

```

---

If suffixed by the letter **f** or **F**, it has type **float**.

855

### Commentary

Unlike its integer counterpart, **short**, there is a suffix for denoting a floating constant of type **float**. Hexadecimal floating constants require the *p* so that any trailing *f* is interpreted as the type **float** suffix rather than another hexadecimal digit.

### Other Languages

Java supports the suffixes *f*, or *F*, to indicate type **float** (the default type for floating constants).

### Coding Guidelines

What is the developers' intent when giving a floating constant a suffix denoting that it has type **float**? The type of a floating constant may not affect the evaluation type of an expression in which it is an operand (which is controlled by the value of the `FLT_EVAL_METHOD` macro). Even the value of the floating constant itself may be held to greater precision than the type **float**.

---

If suffixed by the letter **l** or **L**, it has type **long double**.

856

### Commentary

If greater precision, or range of exponent than type **double**, is required in a floating constant, using the type **long double** suffix may offer a solution. However, possible interaction value of the `FLT_EVAL_METHOD` macro needs to be taken into account.

### Coding Guidelines

Using the **l** suffix in a floating constant may lead to confusion with the digit **1**.

Cg 856.1

If a *floating-suffix* is required only the forms *F* or *L* shall be used.

---

Floating constants are converted to internal format as if at translation-time.

857

### Commentary

The key phrase here is *as if*. Translators wishing to maintain a degree of host independence can still translate to some intermediate form (which is converted to host-specific format as late as program startup). However, all the implications of the conversion must occur at translation-time, not during program execution or startup.

### C90

No such requirement is explicitly specified in the C90 Standard.

In C99 floating constants may convert to more range and precision than is indicated by their type; that is, `0.1f` may be represented as if it had been written `0.1L`.

floating constant  
internal format

`FLT_EVAL_METHOD`

**C++**

Like C90, there is no such requirement in the C++ Standard.

**Common Implementations**

There are a few implementations that use an intermediate form to represent floating constants, independent of the final host on which a program image will execute. The intent is to create a degree of software portability at a lower level than the source code; for instance, implementations that interpret source code that has been translated into the instructions of some abstract machine.

- 
- 858 The conversion of a floating constant shall not raise an exceptional condition or a floating-point exception at execution time.

floating constant conversion not raise exception

**Commentary**

This is a requirement on the implementation. A floating constant that is outside the representable range of floating-point values cannot cause an execution-time exception to be raised. Whether a translator issues a diagnostic if it encounters a floating constant that, had it occurred during program execution, would have raised an exception is a quality-of-implementation issue.

**C90**

No such requirement was explicitly specified in the C90 Standard.

**C++**

Like C90, there is no such requirement in the C++ Standard.

**Recommended practice**

- 
- 859 The implementation should produce a diagnostic message if a hexadecimal constant cannot be represented exactly in its evaluation format;

hexadecimal constant not represented exactly

**Commentary**

The issue of exact representation of floating constants is not unique to hexadecimal notation, but the intent of this notation could lead developers to expect that an exact representation will always be used. An inexact representation can occur if `FLT_RADIX` macro has a value that is not a power of 2, or the hexadecimal floating constant contains more digits than are representable in the significand used by the implementation for that floating type. This is a quality-of-implementation issue, one of the many constructs that a quality implementation might be expected to diagnose. However, the floating-point contingent on the Committee is strong and hexadecimal constants are a new construct, so we have a recommended practice.

852 floating constant representable value chosen

`FLT_RADIX`

**C90**

Recommended practices are new in C99, as are hexadecimal floating constants.

**C++**

The C++ Standard does not specify support for hexadecimal floating constants.

**Coding Guidelines**

A hexadecimal floating constant is a notation intended to be used to provide a mechanism for exactly representing floating-point values. A source file may contain a constant that is not exactly representable. However, support for hexadecimal floating constant notation is new in C99 and at the time of this writing insufficient experience on their use is available to know if any guideline recommendation is worthwhile.

**Example**

```
1 float f = 0x1.11111111111111111111111111111111111111111111111111111111111111111111p11;
```

---

the implementation should then proceed with the translation of the program.

860

### Commentary

Thanks very much.

---

The translation-time conversion of floating constants should match the execution-time conversion of character strings by library functions, such as `strtod`, given matching inputs suitable for both conversions, the same result format, and default execution-time rounding.<sup>64)</sup>

861

### Commentary

The C library contains many of the support functions needed to write a translator. Making these functions available in the C library is just as much about allowing implementation vendors to recycle their code as allowing behaviors to be duplicated. If the translator and its library both use identical functions for performing numeric conversions, there is likely to be consistent behavior between the two environments. The function `strtod` is an example of one such case. However, its behavior will only be the same if the various floating-point mode flags are also the same in both environments.

### C90

This recommendation is new in C99.

### C++

No such requirement is explicitly specified in the C++ Standard.

### Other Languages

A common characteristic of many language translators is that they are written in the language they translate, even Cobol. If a translator is not written in its own language, the most common implementation language is C. Languages invariably specify functionality that performs conversions between character strings and floating-point values. However, it is their I/O mechanisms that perform this conversion and the generated character strings, or floating-point values, are not always otherwise available to an executing program.

Fortran has a large number of numeric functions in its library as does Java. Both of these languages have an established practice of writing their libraries in their respective languages. Although in the case of Fortran, there is also a history of using functions written in C. Java defines conversion functions in `java.lang.float` and `java.lang.double`.

### Common Implementations

The majority of C translators are written in C and call the identical functions to those provided in their runtime library.

### Coding Guidelines

Floating-point values can appear during program execution through two possible routes. They can be part of the character sequence that is translated to form the program image, or they can be read in as character sequences from a stream and converted using the functions defined in Clause 7.20.1.3. In a freestanding environment the host processor floating-point support is likely to be different (it may be implemented via calls to internal implementation library functions) from that available during translation.

It is possible for the original source of the floating-point numbers to be the same; for instance, a file containing a comma separated list of values and this file being **#included** at translation time and read during program execution.

Rev 861.1

A program shall not depend on the value of a floating constant appearing in the source code being equal to the value returned by a call to `strtod` with the same sequence of characters as its first argument.

**Example**

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 #define FLT_CONSTANT_EQ_FLT_STRING(x) (x == strtod(#x, NULL))
5
6 void f(void)
7 {
8     if (!FLT_CONSTANT_EQ_FLT_STRING(1.2345))
9         printf("This translator may not be implemented using its own library\n");
10 }
```

---

862 64) The specification for the library functions recommends more accurate conversion than required for floating constants (see 7.20.1.3).

footnote  
64**Commentary**

Clause 7.20.1.3 describes the `strtod`, `strtof`, and `strtold` functions.

**C++**

There observation is not made in the C++ Standard. The C++ Standard includes the C library by reference, so by implication this statement is also true in C++.

## References

1. ARM. *ARM Developer Suite: Compilers and Libraries Guide*. ARM Limited, 1.2 edition, Nov. 2001.
2. W. D. Clinger. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 92–101, June 1990.