SC 22/WG 14/N2166

Title: Evaluation Formats
Author: Willem Wakker
Date: September 2017

Problem summary

1. The concept of 'evaluation format' is not well (if at all) defined in C11; the only description of the concept is in 5.2.4.2.2p9. As this concept influences the language (it has direct consequence for the effect of the usual arithmetic conversions and possibly overrides the effect of floating suffixes) it should be properly defined in Clause 6 - *Language* rather than being hidden in Clause 5 – *Environment*.

2. 5.2.4.2.2p9 reads (C11, might be changed by DR 500):

   > Except for assignment and cast (which remove all extra range and precision), the values yielded by operators with floating operands and values subject to the usual arithmetic conversions and of floating constants are evaluated to a format whose range and precision may be greater than required by the type.

   The question is: to what circumstances does the word *may* refer? When does this happen and when not? Does this refer to the various values of FLT_EVAL_METHOD or, for instance, to different types of expressions? This needs to be clarified.

3. The above sentence from 5.2.4.2.2p9 includes floating constants. The implied effect of this is that, for instance, **`float`** constants like **`0.3f`** are silently interpreted as **`0.3l`**. This effect is not mentioned or referred to in 6.4.4.2 where the syntax of the floating-suffix is defined, thereby leaving the programmer in the dark. The reason to include floating constants here seems to be: 'constants with more range and precision gives better results'. This 'the compiler knows better' approach might 'help' sloppy or ignorant programmers but punishes the precise programmer who now has to write **`(float)0.3f`** to get what he wants. This all contradicts one of the basic principles of the C language: 'Trust the programmer'! Furthermore it seems that the C++ standard (being even less clear on this issue than the C standard) does not include floating constants in the wider evaluation format, so there is an incompatibility between C and C++ on this point.It is proposed to remove floating constants from the definition of the notion 'evaluation format'.

   This is a 'non-silent' change, causing programs that rely on this automatic widening to behave differently. But I cannot imagine a well-written C program that would be dependent on this feature while I know of several well-written pre-C99 C programs that behave differently (or even incorrect) as a result of this wide evaluation approach. So, removal should not cause big problems; see it as the correction of a mis-guided principle introduced in C99.

4. The 1st sentence of 6.2.5 – *Types* reads:

   > The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it.

   This suggests that a **`float`** function returns a (possibly truncated) **`float`** value, and not a value in a possibly wider evaluation format. This contradicts with 6.8.6.4 – *The return statement* that states that 'the value is returned as if by assignment to an object having the return type of the function'; the following note states 'The **`return`** statement is not an assignment' (for overlap reasons) and the general statement 'The representation of floating-point values may have wider range or precision then implied by the type; a cast may be used to remove this extra range and precision'. No link with the 'greater range or precision' from 5.2.4.2.2, just seemingly a disguised loophole to allow for the return of a value wider than required by the type of the function.

The fact that F.6 requires (if `__STDC_IEC_559__` is defined) that function results are converted to the type of the function confuses the matter even further: this strict behavior has nothing to do with IEEE floating point. So the question is: what purpose is served with allowing wide results in 6.8.6.4? And again: C++ does not allow these 'wide return values' so it is causing incompatibilities with no useful purpose.

It is proposed to remove this 'freedom' to return values in a format greater than the format required by the type of the function.

Proposed replacement text

1. Insert new section 6.3.1.9 (the first text between [] is to be removed if floating constants are not included in evaluation formats, item 3 above, the second text between [] is to be removed if return statements are not included in evaluation formats, item 4 above):

    6.3.1.9 **Evaluation formats**

    The values of floating type yielded by operators subject to the usual arithmetic conversions [and the values of floating constants] are evaluated to a format whose range and precision may be greater than required by the type. Such a format is called an *evaluation format*.note) Assignment and cast operators [and return statements] will always yield values in the format corresponding to the required type. The extend to which evaluation formats are used is defined by the value of FLT_EVAL_METHOD (see 5.2.4.2.2).

    note) The evaluation method determines evaluation formats of expressions involving all floating types, not just real types. For example, if **FLT_EVAL_METHOD** is 1, then the product of two **float _Complex** operands is represented in the **double _Complex** format, and its parts are evaluated to **double**.

2. In 5.2.4.2.2#9 replace:

    Except for assignment and cast (which remove all extra range and precision), the values yielded by operators with floating operands and values subject to the usual arithmetic conversions and of floating constants are evaluated to a format whose range and precision may be greater than required by the type. The use of evaluation formats is characterized by the implementation-defined value of **FLT_EVAL_METHOD**:[24]

    with:

    The use of evaluation formats (see 6.3.1.9) is defined by the implementation-defined value of **FLT_EVAL_METHOD**:

    and remove note 24.

3. In 6.6 note 116: replace “characterized” by “defined”

4. At the end of 6.3.1.8p2 add “(see 6.3.1.9)” and remove note 63

5. In 6.5.4#6: replace “(6.3.1.8)” by “(6.3.1.9)”

6. If floating constants remain subject of the evaluation formats (item 3 above is not accepted), add the following sentence at the end of 6.4.4.2#4:

    Floating constants may be evaluated to a wider format with more range and precision (the evaluation format, see 6.3.1.9) than required by the type indicated by the floating suffixnote).

    note) When **FLT_EVAL_METHOD** is 2, this means that `0.3f == 0.3l`; if a constant with range and precision according to the type is required, a cast should be used: `(float)0.3`.

7. If return statements are excluded from evaluation formats (item 4) remove the last sentence of note 160 and section F.6.