Document: WG14/N1258
Date: 2007/09/10
Project: WG14 TR 24732
References: WG14/N1201, WG14/N1216, WG14/N1231, JTC1/N8645
Authors: Rich Peterson, Jim Thomas
Reply to: Rich Peterson <Rich.Peterson@hp.com>

Subject: Unsuffixed floating constants (unresolved problems with TTDT)

Background: WG14/N1216, "Problems with TTDT", pointed out that C99 Annex F contains normative requirements on the conversion of floating constants to IEEE binary representations; and that these C99 requirements would not be met if WG14/N1201's recommended practice of using _Decimal128 for TTDT were followed in an implementation that uses 128-bit IEEE double-extended for long double (which is a common C99 implementation choice). It further points out that an implementation of TTDT using a decimal radix floating-point type of ***any*** precision ***could not*** meet the Annex F requirements on constant expression evaluation because IEEE 754 binary rounding is required by Annex F. And clearly, use of a binary radix floating-point type for TTDT would not meet the 754R requirements for decimal-radix constants and evaluation. Therefore WG14/N1216 recommended that the TTDT construct be removed from the TR. The paper further suggested that a pragma to treat unsuffixed floating constants as having type _Decimal64 could help make unsuffixed floating constants more usable in programs that use the new decimal types, without breaking programs that rely on behaviors guaranteed by C99 Annex F.

When N1216 was discussed at the London meeting, there was consensus that the problems it pointed out in the recommended practice for TTDT needed to be addressed. But the solution suggested in N1216, of removing TTDT entirely and adding a #pragma to give unsuffixed floating constants type _Decimal64, was not accepted - primarily because a pragma would be a red flag in terms of C++ compatibility. The issue was captured in the minutes (WG14/N1231) as "*Lot of discussion; does not look like consensus to remove TTDT. Soften the language to not require breakage (as it does now) when mixing unsuffixed floating point constants. Using a #pragma will lead to issues with C++*". And there was an action item for Plauger and Peterson to try to come up with words that would do this "softening" such that it that would preserve some of the convenience of TTDT to users of decimal floating-point type without breaking C99 Annex F code and without introducing a #pragma.

However, this action item was not completed before document JTC1/N8645 went out for ballot containing the same TTDT description that the committee had agreed was broken. Since there are multiple ongoing implementations of decimal floating-point that intend to track the TR, we feel it is important to resolve this issue such that the TR to be published does not specify behavior that conflicts with C99 Annex F and the committee agrees is broken. Note that while C99 Annex F is only "conditionally normative", implementations that are concerned about producing floating-point behaviors that are maximally consistent with IEEE 754 are the ones that would specify the Annex as being normative. And those same implementations are among those most likely to be adding support for IEEE 754R decimal floating-point.

Fundamental problems with TTDT

At the meeting, Kwok defended the TTDT feature as described in the TR, on the basis that _Decimal128 was only mentioned in "Recommended practice", and that an implementation could choose to use strings to implement TTDT. However, the concept summary in 7.1.1 does state that TTDT is intended to be an implementation-defined arithmetic type for which all arithmetic operations are defined. It immediately should be troubling that the support for the 754R types would include arithmetic that is totally implementation-defined. N1216 showed that none of the decimal-radix floating-point types would be sufficient to represent long double constants in the common case of IEEE double-extended precision; no decimal radix type can meet Annex F

requirements on translation-time constant arithmetic, and no binary-radix floating-point type can satisfy the requirements for decimal radix constants (even if precision were not an issue, the quantum exponent would not be preserved). So using an arithmetic type for TTDT isn't really a valid choice. That leaves only string representation.

But even using strings to implement TTDT, the TR clearly intends that translation-time evaluation of constant expressions in the TTDT type can be performed with "no 'top-down' parsing context information". The quote is from item 4 in 7.1.1 in regard to processing individual constants, but the second example below it shows that a compiler would be permitted to evaluate an expression containing just TTDT constants without context. Whatever an implementation did to implement arithmetic on the string representation (including infinite precision arithmetic as in Ada, which was also suggested at the meeting), it couldn't satisfy both the Annex F requirements for binary rounding and also the requirements for decimal rounding, because they are different! Similarly for producing results consistent with both FLT_EVAL_METHOD and DEC_EVAL_METHOD.

Even if it weren't for specific conflicts with Annex F requirements, the TTDT constant evaluation scheme in the TR has the highly undesirable property that it would change the observed behavior of many existing strictly-conforming C99 programs: TTDT evaluation applies to all unsuffixed (non-hexadecimal) floating constants, which surely comprise the majority of floating constants used in real applications. And even if all such behavior changes might be within the range of variation allowed under C99 without Annex F, as a practical matter large mature applications and their test systems often come to rely upon specific floating-point results on a given platform. Sometimes changes in compiler optimization cause differences in test results, and those differences need to be understood and resolved by the owners of such applications either by changing the source or by deciding that the new result is acceptable. It would come as a very unwelcome surprise to the owners of such applications if the addition of the decimal floating type feature to the compiler caused a large number of differences in the test results of their mature C99 (or C90) application! The TTDT evaluation currently in the TR violates the "principle of least astonishment" by changing the behavior of code that predates the existence of decimal floating types – especially so given that the types are specified such that they do not otherwise interact with the C99 floating types, e.g. by participating with them in the usual arithmetic conversions.

Below are three kinds of alternatives for how the TR could specify the treatment of unsuffixed floating constants in a way that would not imply any change in the behavior of C99 code. They are listed in order of increasing complexity. And within each general kind of alternative, there are several options for how certain details might be handled.

Alternative #1: Eliminate TTDT: unsuffixed floating constants have type double

Although considered and rejected previously, this deserves reconsideration in the light of how problematic TTDT as proposed really is, and how complex it appears to be to fix it. TTDT is primarily a convenience/cosmetic feature for novices using the new decimal types introduced in the TR. With unsuffixed constants always treated as type double, and no implicit conversions, programmers who fail to put a suffix on constants used in expressions with decimal operands would often get a diagnostic from the compiler. However, in the important special cases of using an unsuffixed constant to initialize or assign to an object of decimal floating type, the constant would be treated as type double and then silently converted to decimal type, potentially losing the exact value as well as the quantum exponent that would be preserved if the constant were treated as having decimal type.

The TR already says in Section 7:

> Change 6.4.4.2#4 to:
> [4] An unsuffixed floating constant has type **`double`**. If suffixed by the letter **`f`** or **`F`**, it has type **`float`**. If suffixed by the letter **`d`** or **`D`**, it has type **`double`**. If suffixed by the letter **`l`** or **`L`**, it has

type **long double**.

Option #1a
The change to the TR would simply be to remove section 7.1 in its entirety.

Option #1b
Like #1a but also make mixing generic and decimal floating types a constraint violation in simple assignment (which covers initialization), return statement, and argument-passing. These contexts do not share the technical problem of determining an appropriate common type to which to convert mixed radix operands in an arithmetic expression. However, a programmer using decimal types in an application where the values are such that the arithmetic ought to produce results exact to the penny with meaningful quantum exponents would be equally puzzled by bugs caused by treatment of the constant 2.01 as type double instead of a decimal floating type, no matter whether the constant was being added to a variable of decimal floating type or assigned to one (or written on the return statement of a decimal floating-point function, or passed as an argument to a function with a decimal floating-point parameter).

Option #1c
Like #1b, but make the constraint only for assignment/return/argument-passing where a generic floating type is converted to a decimal floating type, but not when converting decimal floating to generic floating. This reflects the fact that the problem being addressed is accidental use of an unsuffixed constant being treated as type double in a decimal context – accidentally using a decimal value in these three contexts when they expect a generic floating-point type is not nearly as likely or as harmful.

Alternative #2: Eliminate TTDT: the type of unsuffixed constants is a translation-time option

In the absence of something like TTDT, the most obvious way to control the behavior of unsuffixed floating constants such that they could be given a decimal floating-point type instead of type double would be via some sort of compiler directive. The C99 mechanism for this is the #pragma directive or, equivalently, the _Pragma operator. Given that at the London meeting the idea of using a #pragma was rejected primarily because it would be a red flag to C++ compatibility, other alternatives for directives might be considered.

Option #2a
C++ is considering adding some sort of attribute syntax to the language, and C is also considering it for a future revision. The TR could define a #pragma to control the behavior based on C99, making it clear that this could also be also be specified with attribute syntax when available in the language. As a concrete proposal:

```
#pragma STDC ATTRIBUTE_UNSUFFIXED_FLOAT_IS_DECIMAL64
```

Option #2b
Another way to avoid the C++ issue with #pragma might be to make it a translation-time option without specifying the mechanism to control it. While the standard doesn't have a way to mandate translation-time options, it is common for predefined macros to reflect features of the implementation that in practice are subject to translation-time options. So while less flexible than a #pragma an alternative might be simply to state:

> An implementation may provide a translation-time option to give unsuffixed decimal floating constants type _Decimal64 instead of type double. If an implementation is giving unsuffixed decimal floating constants type _Decimal64 within a translation unit, it must predefine the macro __STDC_UNSUFFIXED_FLOAT_IS_DECIMAL64__.

The macro would allow source code to protect itself from the option by adding something like:

```
#if __STDC_UNSUFFIXED_FLOAT_IS_DECIMAL64__
```

```
#error This code expects unsuffixed floating constants to have type double.
#endif
```

But obviously this approach has the problem of needing to change the source of preexisting code to avoid a problem introduced by a change in the implementation.

Alternative #3: Specify TTDT with "as if" rules that provide context

The TTDT concept could specify behavior with "as if" rules treating unsuffixed floating constants as being represented internally as strings with the exact spelling seen in the source code, and conversion of those strings to floating types, and evaluation of expressions containing them, deferred until the entire containing expression had been seen. This would prevent changes to the behavior of C99 code, and could provide even more novice-friendly behavior in decimal floating-point contexts, without mandating specific implementation choices. Below is an attempt at rewriting section 7.1.1 of the TR to do this:

Option #3a

7.1.1 Translation time data type

In order to allow code to use unsuffixed decimal floating constants naturally with either generic floating types or decimal floating types, the concept of a translation time data type (TTDT) is introduced. The purpose of TTDT is to defer the choice of floating-point representation for the value of an unsuffixed constant until the context of its use is known. In contexts where there are no uses of decimal floating types, the TTDT is given type double. This ensures that existing C99 programs (which by definition cannot use decimal floating types) will experience absolutely no change in behavior caused by the introduction of the decimal floating types. In contexts where decimal floating types are used, the TTDT is given the type of the widest decimal floating type used in that context. This maximizes precision and preservation of the quantum exponent without excessive use of _Decimal128.

The problem is specifying the context and behavior in a way that will produce the same result in all implementations, in an implementation-neutral manner. We rely on "as if" rules to do this.

Suggested changes to C99:

In 6.2.5 after paragraph 28, add a paragraph:

[28a] The translator uses an internal representation called the translation time data type, or TTDT, to handle unsuffixed floating constants. TTDT is the type initially given to an unsuffixed decimal floating constant, before the context of its use can determine whether it should be of type double or one of the decimal floating-point types. It behaves as if it simply captures the string represention of the constant directly from the source code. It is not checked for range or precision, as that can only be done after its floating-point type is determined. There is no type specifier for TTDT, it is an internal translation-time artifact.

Replace 6.4.4.2 paragraph 4 with the following:

[4] An unsuffixed floating constant has type TTDT if it is decimal, or type double if it is hexadecimal. If suffixed by the letter f or F, it has type float. If suffixed by the letter d or D, it has type double. If suffixed by the letter l or L, it has type long double.

After 6.5 Expressions paragraph 8, add a new paragraph:

[8a] An expression containing a value of type TTDT cannot be evaluated until the *floating-point-complete* expression containing it is available for evaluation. When a value of type TTDT is encountered as an operand in an expression, the operator to which it is an operand is marked "contains TTDT", and the expression is left unevaluated. When an expression is *floating-point-complete*, if it is marked "contains TTDT" then the behavior is as if that complete expression were scanned to determine the actual type representation for TTDT as the largest decimal floating-point

type of any operand within it, or else type double if the expression contains no decimal floating-point operands. Then finally the expression can be evaluated in the usual way, changing each TTDT value to the floating-point type determined in the previous scan. The new type is established not by arithmetic conversion, but rather as if by appending the appropriate floating-suffix to the source representation held in the TTDT. An implementation need not perform separate scans to save complete expressions involving TTDT values, determine the type to be used for TTDT, and then evaluate the complete expression applying lexical edits to the source representation of each TTDT, as long as the result is the same as if these scans were performed. A *floating-point-complete* expression is defined to be any of the following:

- the expression in an expression statement;
- the controlling expression of a selection statement;
- any of the expressions of an iteration statement;
- the expression in a return statement;
- the operand of the sizeof operator;
- an initializer;
- a compound literal;
- a case label;
- an argument to a function call;
- a cast;
- the left operand of a comma operator;
- the first operand of the ternary operator;
- the operand of the "!" operator;
- either operand of the "||" or "&&" operators.

Option #3b
Like #3a, except that if decimal types are encountered during the scan, instead of giving each TTDT the largest of the decimal types encountered, give each TTDT the smallest decimal type capable of representing its value with full precision and quantum exponent as implied by its spelling. This could result in constant expressions losing precision where they would not under option #3a.

Option #3c
Like #3b, but instead of giving each TTDT a decimal type that depends on its value and spelling, give it the type _Decimal64. Again, this could result in precision loss that would not happen under #3a, although the precision loss would be analogous to what happens in C99 when unsuffixed constants are given type double and used in expressions that later are promoted to type long double.

Option #3d
Other rules for defining the context used to determine whether TTDTs should be treated as type double or a decimal floating-point type. E.g. instead of the broad context of "floating-point-complete", the context could be the nearest containing expression that has any floating-point type.

Option #3e
Make it a constraint violation if both generic floating-point types and decimal floating-point types appear within the expression context used to determine the type of a TTDT.