Rationale for TR 24732

Extension to the programming language C

Decimal Floating-Point Arithmetic

Contents

1 Introduction

1.1 Background

The existing floating-point types in the C language are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic; the standard does not require the floating-point types to be a specific representation or radix. For this Technical Report, the committee considered both adding decimal floating-point support without introducing additional data types, as well as the current proposal of adding three new types (as per IEEE-754R) to the language.

Most applications do not care how floating-point is done. Many applications would be better off using decimal floating-point. Very few applications need the better error bounds of binary floating-point. There will be applications that will need both kinds of floating-point (many will be conversion programs used to convert existing data files from binary floating-point to decimal floating-point). There will be a few applications that will need to run a mixture of third party libraries that only know about binary floating-point, and other third party libraries that only know about decimal floating-point.

Binary floating-point and decimal floating-point (as defined in IEEE-754R) occupy the same amount of storage, and they could be treated the same for all data movement and register usage. This means that a function call whose prototype is binary floating-point, but is called with decimal floating-point (and visa versa), can be made to work (as the same number of bytes are passed in the same manner). Hence, adding several functions to the library to convert between binary floating-point and decimal floating-point (for the same sized data) would allow applications to mix both kinds of floating-point. Of course, this means that the application needs to add explicit function calls to do the conversions.

The floating-point unit (FPU) does a binary float operation versus a decimal float operation by either a different opcode, or by a switchable mode bit in some control word. In either case, code generation must be controllable by the user. Switching the mode bit at runtime could be done by a function call. But, generating different opcodes require translation time control - a pragma seems like the logical choice; this also works for switching the mode bit.

Based on the above, one might come to the conclusion that adding decimal floating-point support to the language can be done by reusing the existing floating-point types, with some combination of compiler switch, pragma, and conversion routines to enable a mixed binary/decimal floating-point operations. This approach, however, does present several problems.

Variable argument functions do float to double promotion. This will be incorrect if the hardware promotes as if the data is binary, but the data is really decimal, and visa versa. Explicit calls to some conversion routines would make it work; however, it would be cumbersome to use.

Debugging tools would have no clue if a floating-point object is decimal or binary. That is, a float, double, or long double declaration does not imply the base that will be used for that object. In fact,

the object could be binary floating-point some places in the program and decimal floating-point in others.

By introducing three additional floating-point data types to the language resolves some of these issues. However, adding new data types can be seen as making the language, as well as their use alongside the existing floating-point data types, unnecessarily complex. Some arguments presented for having separate decimal floating-point types are:

1. The fact that there are two sets of floating-point types in itself does not mean the language would become more complex. The complexity question should be answered from the perspective of the user's program; that is, do the new data types add complexity to the user's code? The answer is probably no except for the issues surrounding implicit conversions. For a program that uses only binary floating-point types, or uses only decimal floating-point types, the programmer is still working with three floating-point types. Having additional data types is not making the program more difficult to write, understand, or maintain.

2. Implicit conversions, other than assignment and function argument passing, can be handled by simply disallowing them (except maybe for cases that involve literals). If we do this, for programs that have both binary and decimal floating-point types, the code is still clean and easy to understand.

3. If we only have one set of data types, and if we provide STDC pragmas to allow programs to use both representations, in a large source file with STDC pragma changing the meaning of the types back and forth, the code is actually a field of land mines for the maintenance programmer, who might not immediately aware of the context of the piece of code.

   Since the effect of a pragma is a lexical region within the program, additional debugger information is needed to keep track of the changing meaning of data types.

4. Giving two meanings to one data type hurts type safety. A program may bind by mistake to the wrong library, causing runtime errors that are difficult to trace. It is always preferable to detect errors during compile time. Overloading the meaning of a data type makes the language more complicated, not simpler.

5. A related advantage of using separate types is that it facilitates the use of source checking/scanning utilities (or scripts). They can easily detect which floating-point types are used in a piece of code with just local processing. If a STDC pragma can change the representation of a type, the use of grep, for example, as an aid to understand and to search program text would become very difficult.

6. Suppose the standard only defines a library for basic arithmetic operations. A C program would have to code an expression by breaking it down into individual function calls. This coding style is error prone, and the resulting code difficult to understand and maintain. A C++ programmer would almost definitely provide his/her own overloaded operators.

> Rather than having everyone to come up their own, we should define it in the standard. If C++ defines these types as class, C should provide a set of types matching the behavior.

This is not a technical issue for an implementation, as it might seem on the surface initially - that is, it might seem easier to just provide new meaning to existing types using a compiler switch - but is an issue about usability for the programmer. The meaning of a piece of code can become obscure if we reuse the float/double/long double types. Also, we have a chance here to bind the C behavior directly with IEEE, reducing the number of variations among implementations. This would help programmer writing portable code, with one source tree building on multiple platforms. Using a new set of data types is the cleanest way to achieve this.

1.2 The Arithmetic Model

Based on a model of decimal arithmetic[1], which is a formalization of the decimal system of numeration (Algorism), as further defined and constrained by the relevant standards: IEEE-854, ANSI X3-274, and the proposed revision of IEEE-754. The latter is also known as IEEE-754R.

1.3 The Encodings

Based on the current IEEE-754R proposal.

C99 specifies floating-point arithmetic using a two-layer organization. The first layer provides a specification using an abstract model. The representation of floating-point number is specified in an abstract form where the constituent components of the representation is defined (sign, exponent, significand) but not the internals of these components. In particular, the exponent range, significand size and the base (or radix), are implementation defined. This allows flexibility for an implementation to take advantage of its underlying hardware architecture. Furthermore, certain behaviors of operations are also implementation defined, for example in the area of handling of special numbers and in exceptions.

The reason for this approach is historical. At the time when C was first standardized, there was already various hardware implementations of floating-point arithmetic in common use. Specifying the exact details of a representation would make most of the existing implementations at the time not conforming.

C99 provides a binding to IEEE-754 by specifying an annex F and adopting that standard by reference. An implementation not conforming to IEEE-754 can choose to do so by not defining the macro __STDC_IEC_559__. This means not all implementations need to support IEEE-754, and the floating-point arithmetic need not be binary.

The technical report specifies decimal floating-point arithmetic according to the IEEE-754R, with the constituent components of the representation defined. This is more stringent than the existing

[1] A description of the arithmetic model can be found in http://www2.hursley.ibm.com/decimal/decarith.html.

C99 approach for the floating types. Since it is expected that all decimal floating-point hardware implementations will conform to the revised IEEE 754, binding to this standard directly benefits both implementers and programmers.

2 General

2.1 Scope

2.2 References

3 Predefined macro name

4 Decimal floating types

The three new decimal floating-point data types introduced in the technical report have names similar and characteristics matching those defined in IEEE-754R. An alternative naming convention that encapsulates the base (or radix) and precision in the name had also been suggested; for example: decfp7, decfp16, and decfp34, which indicate decimal representation (dec), floating point type (fp), with the specified number of coefficient digits (7, 16, or 34). However, it was felt that names similar to those used in IEEE-754R may be more appropriate.

Also a single token used as a type name would make it easy for C++ to implement the types as classes.

Decimal floating types are distinct types from the real floating types *float*, *double*, and *long double*, even if an implementation chooses the same decimal representation for the real floating types.

The technical report does not specify decimal complex nor decimal imaginary types; however, this does not mean that they can not be added in the future.

5 Characteristics of decimal floating types <decfloat.h>

A new header is introduced that defines the characteristics for the new decimal floating types. It defines a set of macros similar to the ones defined for real floating types in <float.h>.

6 Conversions

6.1 Conversions between decimal floating and integer

When the new type is a decimal floating type, we have these choices: the most positive/negative number representable, positive/negative infinity, and quiet NaN. The first provides no indication to the program that something exceptional has happened. The second provides indication, and since other operations that produce infinity also raise exception, an exception would be raised here for consistency. The third allows the program to detect the condition and provides a way for the implementation to encode the condition (for example, where it occurs).

When the new type is an unsigned integral type, the values that create problems are those less than 0 and those greater than U*type*_MAX. There is no overflow/under-flow processing for unsigned arithmetic. A possible choice for the result would be U*type*_MAX if the original value is positive, or 0 if negative. Also, implementations are not required to raise signals for signed integer arithmetic. When the new type is a signed integral type, the values that create problems are those less than *type*_MIN and those greater than *type*_MAX. The result here should be *type*_MIN or *type*_MAX depending on whether the original value is negative or positive.

Conversions between decimal floating and integer formats follow the operation rules as defined in IEEE-754R.

6.2 Conversions among decimal floating types, and between decimal floating types and generic floating types

The specification is similar to the existing ones for float, double and long double, except that when the result cannot be represented exactly, the behavior is defined to become correctly rounded.

6.3 Conversions between decimal floating and complex

When a value of decimal floating type is converted to a complex type, the real part of the complex result value is determined by the rules of conversion in 6.2 and the imaginary part of the complex result value is zero.

6.4 Usual arithmetic conversions

In a business application that is written using decimal arithmetic, mixed operations between decimal and other real types might not occur frequently. Situations where this might occur are when interfacing with other languages, calling an existing library written in binary floating-point arithmetic, or accessing existing data. The programmer may want to use an explicit cast to control

the behavior in such cases to make the code maximally portable. One way to handle usual arithmetic conversion is therefore to disallow mixed operations. The disadvantage of this approach is usability - for example, it could be tedious to add explicit casts in assignments and in function calls when the compiler can correctly handle such situations. A variation of this is to allow it only in simple assignments and in argument passing.

One major difficulty of allowing mixed operation is in the determination of the common type. C99 does not specify exactly the range and precision of the generic real types. The pecking order between them and the decimal types is therefore unspecified. Given two (or more) mixed type operands, there is no simple rule to define a common type that would guarantee portability in general.

For example, we can define the common type to be the one with greater range. But since a double type may have different range under different implementations, a program cannot assume the resulting type of an addition, say, involving both _Decimal64 and double. This imposes limitations on how to write portable programs.

If the generic real type is a type defined in IEEE-754R, and if we use the *greater-range rule*, the common type is easily determined. When mixing decimal and binary types of the same type size, decimal type is the common type. When mixing types of different sizes, the common type is the one with larger size. The suggested change in Annex A uses this approach but does not assume the generic real type to follow IEEE-754R. This guarantees consistent behaviors among implementation that uses IEEE-754 in their binary floating-point arithmetic, and at the same time provides reasonable behavior for those that don't.

The committee felt that few programs will require mixed operations, and that requiring explicit cast may result in less error-prone programs.

6.5 Default argument promotion

There is no default argument promotion specified for the decimal floating types in the technical report.

7 Constants

New suffixes are introduced to denote decimal floating constants. Also, the **d** and **D** suffixes are added to denote type double.

7.1 Unsuffixed decimal floating constant

The proposal for a translation-time data type (TTDT) to allow for the use of unsuffixed floating-point constants originated in WG14 paper N1108. At the Lillehammer meeting, the committee felt

that the idea was too important to leave out, and as a minimum it should be a recommended practice in this technical report. There were extensive discussions on whether TTDT should be made part of the rules, i.e. 'required'. In the end the committee decided to make it a separate section in the TR. Note also TTDT could apply to TR 18037.

7.1.1 Translation time data type

Translation time data type (TTDT) is an abstract data type which the translator uses as the type for unsuffixed floating constants. A floating constant is kept in this type and representation until an operation requires it to be converted to an actual type. The value of the constant remains exact for as long as possible during the translation process. The concept can be summarized as follows:

1. The implementation is allowed to use a type different from double and long double as the type of unsuffixed floating constant. This is an implementation defined type. The intention is that this type can represent the constant exactly if the number of decimal digits is within an implementation specified limit. For an implementation that supports decimal floating pointing, a possible choice is the widest decimal floating type.
2. The range and precision of this type are implementation defined and are fixed throughout the program.
3. TTDT is an arithmetic type. All arithmetic operations are defined for this type.
4. Usual arithmetic conversion is extended to handle mixed operations between TTDT and other types. If an operation involves both TTDT and an actual type, the TTDT is converted to an actual type before the operation. There is no "top-down" parsing context information required to process unsuffixed floating constants. Technically speaking, there is no deferring in determining the type of the constant.

Examples:

```
double f;
f = 0.1;
```

Suppose the implementation uses _Decimal128 as the TTDT. 0.1 is represented exactly after the constant is scanned. It is then converted to double in the assignment operator.

```
f = 0.1 * 0.3;
```

Here, both 0.1 and 0.3 are represented in TTDT. If the compiler evaluates the expression during translation time, it would be done using TTDT, and the result would be TTDT. This is then converted to double before the assignment. If the compiler generates code to evaluate the expression during execution time, both 0.1 and 0.3 would be converted to double before the multiply. The result of the former would be different but more precise than the latter.

```
float g = 0.3f;
f = 0.1 * g;
```

When one operand is a TTDT and the other is one of float/double/long double, the TTDT is converted to double with an internal representation following the specification of FLT_EVAL_METHOD for constant of type double. Usual arithmetic conversion is then applied to the resulting operands.

```
_Decimal32 h = 0.1;
```

If one operand is a TTDT and the other a decimal floating type, the TTDT is converted to _Decimal64 with an internal representation specified by DEC_EVAL_METHOD. Usual arithmetic conversion is then applied.

If one operand is a TTDT and the other a fixed point type, the TTDT is converted to the fixed point type. If the implementation supports fixed point type, it is a recommended practice that the implementation chooses a representation for TTDT that can represent floating and fixed point constants exactly, subjected to a predefined limit on the number of decimal digits.

8 Floating-point environment <fenv.h>

[This will likely be renamed to a new <decfenv.h> header to isolate the new additions and to avoid namespace pollution. Alternatively we could introduce a macro to control the inclusion of the new library – require committee discussions.]

The new, unique rounding mode FE_DEC_TONEARESTFROMZERO for decimal floating-point operations corresponds to the IEEE-754R rounding mode “Round to Nearest, Ties Away from Zero”.

8.1 The DEC_MAX_PRECISION pragma

Certain algorithms or legal requirements may stipulate a precision on the result of an operation; and this precision could be different from those of the three standard types. This pragma changes the precision that would be used for all decimal floating-point operations.

[Should this pragma apply to DFP constants? Why/why not? – require committee discussions.]

9 Arithmetic Operations

9.1 Operators

9.2 Functions

9.3 Conversions

10 Library

10.1 Decimal mathematics <math.h>

The list of elementary functions specified in the mathematics library is extended to handle decimal floating-point types.

[The new functions should be moved to a new <decmath.h> header to isolate the changes and to avoid namespace pollution. Alternatively we could introduce a macro to control the inclusion of the new library – require committee discussions.]

10.2 New functions

IEEE-754R specifies two additional decimal floating-point operations: *samequantum* and *quantize*. These are implemented as new library functions in C99. The library functions have the same semantics as the IEEE operations.

10.3 Formatted input/output specifiers

New length modifiers are introduced for decimal floating types.

10.4 strtod32, strtod64, and strtod128 functions <stdlib.h>

The latest IEEE-754R draft requires that floating-point overflow be raised for values that are too large or too small. As such, setting errno to ERANGE as currently proposed does not meet those requirements (but does match strtod family). Perhaps the requirements of 7.12.1#4 of math_errhandling should be applied to the strto* functions. [require committee discussion.]

10.5 wcstod32, wcstod64, and wcstod128 functions <wchar.h>

10.6 Type-generic macros <tgmath.h>

Annex A