This document proposes to add decimal floating point support to the C++ standard. The current version doesn’t spell out the details but instead refers to the Decimal TR (ISO/IEC TR 24733) as a basis and describes changes to be applied to this interface to bring the proposal up to date with C++ 2011 enhancements.

**Introduction**

C++ provides built-in data types for the processing of numerical values: float, double, and long double. The constraints for these types imply that a floating point representation is used, i.e., the values are represented using a fixed size as \((-1)^{sign} \times significand \times base^{exponent}\). The standard doesn’t mandate the base to be used and typically base 2 is chosen primarily because it yields the fastest computations.

In many areas, especially in finance, exact values need to be processed and the inputs are commonly decimal. Unfortunately, decimal values cannot, in general, be represented accurately using binary floating points even when the decimal values only uses a few digits. Instead, the values become an approximation. As long as the values are carefully processed the original decimal value can be restored from a binary floating point (assuming reasonable restrictions on the number of decimal digits). However, computations and certain conversions introduce subtle errors (e.g. double to float and back to double, even if float is big enough to restore the original decimal value). As a result, the processing of exact decimal values using binary floating points is very error prone.

The use of decimal floating points avoids many of the problems caused by binary floating points. In particular, computations which need to accurately process decimal numbers can use decimal floating points. Decimal floating points provide a useful and sufficient compromise for these domains. Since they use a fixed size representation computations which are normally exact can introduce inaccuracies when the number of necessary digits becomes too big but for actual applications this is rarely a problem. Also, decimal floating points cannot represent the result of all operations exactly. For example, the result of a division with a prime other than 2 and 5 will, in general, be rounded. In the contexts where exact results are needed the corresponding operations aren’t needed.
The need for support of exact decimal computations is recognized in many communities and supported in several systems, although different alternatives for the support are chosen. Below is a list of example programming languages with decimal support:

1. The C committee is working on a Decimal TR as TR 24732. The decimal support in C uses built-in types _Decimal32, _Decimal64, and _Decimal128.
2. Java provides decimal arithmetic by java.math.BigDecimal, an arbitrary sized integer with an integer scale for the decimal places.
3. Python provides decimal.Decimal which is a fixed point decimal representation. The number of decimal digits can be set globally.
4. .Net provides System.Decimal which is a 128 bit decimal floating point. The details of this representation are slightly different from the 128 bit decimal floating point in IEEE 754–2008. System.Decimal is accessible in C# as decimal.
5. SQL provides a fixed point decimal representation where the number of digits and the number of fractional digits can be chosen for each context.
6. Ruby provides BigDecimal, an arbitrary sized integer with an integer scale for the decimal places.

Since C++ is used in many places where accurate decimal arithmetic is required it seems reasonable to add similar support to the standard C++ library.

Proposal

This document proposes to add the interfaces described by the Decimal TR, augmented to take advantage of C++ 2011 features as outlined below, as a mandatory part of the next major revision of C++ (currently scheduled for 2017).

The Decimal TR was issued in 2009 and, thus, in 2014 a statement needs to be made whether it is to be affirmed, revised, or withdrawn. Since the next revision of C++ which is open to additions is scheduled to be released in 2017 it is also proposed that the Decimal TR is revised to reflect the changes outlined below for the 2014 systematic review. Assuming decimal floating point support is added to C++ 2017 the technical report can be withdrawn for the 2019 systematic review.

Potential Implications

Adding anything to the standard C++ library isn’t free and any component may depend on specific infrastructure to be present to be implementable. This section discusses the involved costs and requirements.

Hardware Support

The support for decimal floating point numbers described does not require specific hardware support. There are several software implementations of decimal floating points (Intel, IBM, HP) with suitable performance. It isn’t expected that decimal floating points are used for heavy number crunching because in these contexts the corresponding results will not be exact decimal values in the first place. Thus, the performance expectations for decimal floating points are different than those for floating points used for number crunching.
When processing decimal values using binary floating points it is necessary to convert between fractional decimal
to the closest fractional binary value. These conversions are relatively expensive and are avoided when the
processing is done with decimal floating points. Since the operations on binary floating points in general yield
inaccurate decimal values the hardware support for binary floating point isn’t of much help when trying to
process decimal values. Thus, performance comparisons between decimal floating points and binary floating
points are misguided because they address different problems.

That said, dedicated hardware can improve the performance of operations using decimal floating points and the
specification is written such that potentially available hardware support can be used for an implementation. For
example, IBM provides a library detecting the presence of hardware support and which only uses a software
implementation for decimal floating points where no hardware support is available (see the section on DFPAL:
the decimal support in gcc is based on libdfp which also chooses between a hardware and a software
implementation depending on availability of the hardware but which implements the interface of TR 24732).

Cost of Specification

The operations on decimal floating points are relatively complex. To yield predictable results for portable
programs it is necessary to specify the details of rounding, retained precision, dealing with boundary conditions,
etc. However, all of these details are already addressed by IEEE 754–2008. The specification in the C++
standard will have exactly the same semantics by referencing IEEE 754–2008 for the semantics. What needs to
be specified are the interfaces to access the various features of IEEE 754–2008 in a natural way from C++.

The Decimal TR already spells out most aspects of a C++ binding. With the added C++ 2011 features it is
possible to create a better user experience. There are some design areas open with respect to adding C++ 2011
support (see the section on Changes to the Decimal TR below). Thus, the overall cost of specification should be
acceptable.

Cost of Implementation

The implementation of decimal floating point support is certainly not trivial. However, it is also not as complex as,
e.g., the implementation of the special math functions. Several independent implementations are available for
different platforms, including open source versions. The libraries mentioned below are all using a C interface
which can be used to implement the C++ support.

1. Intel’s library is distributed as source.
2. IBM’s implementation is distributed as source with multiple open source projects (gcc and ICU).
3. HP provides support for decimal floating points with their C and C++ compilers.

Implementing the interfaces specified by the Decimal TR in terms of the C implementations is relatively straight
forward. It also seems reasonable that a native C++ implementation can be provided with a reasonable amount
of work.

Cost of Testing

IBM provides a set of language independent test cases for the decimal floating point semantics on the General
Decimal Arithmetic page. These can be processed by a C++ program to yield a reasonable basis for testing. A
comprehensive testsuite for the decimal floating point semantics is probably more involved but such testsuites can be shared with other languages also requiring support for decimal floating point support, e.g., C, ECMA Script, etc. Testing the various C++ interfaces, i.e., the language specific parts which can’t be shared, shouldn’t be more involved than other C++ libraries.

Cost of Support

The semantics of decimal floating points is very similar in spirit to the semantics of binary floating points. The primary difference is that the base is decimal rather than binary. The major difference between binary and decimal floating points is that the latter are not normalized, i.e., individual decimal values may have multiple representations (a group of different representations for the same value is referred to as cohort by IEEE 754–2008). The freedom can be used to keep track of the precision of values and needs to be maintained during rounding. However, the overall complexity of the decimal floating point semantics are on a similar level as those of binary floating points. They are not dramatically more complex as is the case, e.g., with the special math functions. Staff capable of providing support for use of binary floating points will be able to also provide support for decimal floating points. To some extent, using and providing support for decimal floating points is easier than for binary floating points because all issues relating to base conversions disappear.

Changes to the Decimal TR

The Decimal TR was targetting C++ 2003 and, thus, didn’t use any of the new C++ 2011 features. Several of the new features help in creating a better user experience and the specification in the Decimal TR needs to be updated to take these into account. This section describes the changes proposed to the Decimal TR.

Standard Layout Types

C++ 2003 didn’t have any concept of standard-layout types and it was impossible to make declared default constructors trivial to take advantage of POD types. In C++ 2011 the restrictions on types which can be treated special are relaxed and standard-layout types are defined which support types with private non-static data members. Standard-layout types are, e.g., needed when communicating with other language. Thus, all decimal types will be required to be standard-layout types.

Defaulted Default Constructors

To make a decimal type a POD type it needs to be a standard-layout type and a trivial class. Since there are several non-trivial constructors in each of the decimal types it is necessary to declare the default constructor. To keep the class trivial the default constructors need to be defaulted on the first declaration. The corresponding declarations will be changed to become

```cpp
decimal32() = default;
decimal64() = default;
decimal128() = default;
```

The Decimal TR couldn’t make the decimal types trivial because there was no way for C++ 2003 to make an user-declared default constructor trivial. Instead, the Decimal TR defined the default constructor to initialize the decimal floating point with a zero value. For decimal floating points there is a large cohort of zero values and
whichever zero is chosen is unlikely to be the right one in practice. Thus, using an explicitly defaulted default constructor is a semantic change possibly resulting in non-initialized decimal floating points but the advantages of this change seem to outweigh the disadvantages.

**Explicit Conversion Operators**

The decimal floating-point types all have a conversion operator to `long long` obtaining the value truncated towards zero. This conversion yields an unspecified result when the integral part cannot be represented by `long long` or if the decimal type represents one of the special values. Note, that the Decimal TR used the type `long long` although it was introduced only with C++ 2011. This was done to avoid compatibility issues between an implementation based on the TR and an implementation augmented to use new C++ 2011 features.

Although the conversion is sometimes useful it shouldn’t be implicit, i.e., these conversion operators will be made explicit:

```cpp
explicit operator long long() const;
```

The behavior of these conversion operators will remain unchanged. Making the conversion explicit introduces an inconsistency with the existing floating point types `float`, `double`, and `long double`: These can be converted implicitly to integer types. Since the implicit conversions from floating point types to integers frequently introduce surprises it seems to be reasonable to make the conversion explicit for newly introduced types.

Section 4.2 (Conversions) of the Decimal TR describes how decimal floating-point types can be converted to basic floating types using a cast in C. Since implicit conversion between decimal floating-point types and basic floating types can easily create problems corresponding conversions are not available in the Decimal TR. With the possibility of disabling implicit conversions corresponding explicit conversions should be added:

```cpp
explicit operator float() const;

Returns: If `std::numeric_limits<float>::is_iec559 == true`, returns the result of the conversion of *this to float, performed as in IEEE 754–2008. Otherwise, the returned value is implementation-defined.
```

```cpp
explicit operator double() const;

Returns: If `std::numeric_limits<double>::is_iec559 == true`, returns the result of the conversion of *this to double, performed as in IEEE 754–2008. Otherwise, the returned value is implementation-defined.
```

```cpp
explicit operator long double() const;

Returns: If `std::numeric_limits<long double>::is_iec559 == true`, returns the result of the conversion of *this to long double, performed as in IEEE 754–2008. Otherwise, the returned value is implementation-defined.
```

Whether the various decimal* to *() conversion functions used by the current Decimal TR are retained needs to be decided. In some contexts it may be preferrable to use named functions. For example, the conversion operators are not necessarily suitable to be used as function objects. On the other hand, it is easy to create corresponding function objects using the explicit conversions.
Make the Decimal Types final

Although the decimal floating-point types are described as a library feature, some restrictions are imposed on them to allow implementing these types as built-in types. In particular, Section 2 (Conventions) states that the result of deriving from the decimal floating-point types is undefined. Instead of making this behavior undefined, all of the decimal floating-point types should be made final to prevent deriving:

```cpp
class decimal32 final { ... };
class decimal64 final { ... };
class decimal128 final { ... };
```

Use of other operations capable of detecting if the type is implemented as a class or is a built-in type will remain undefined.

Exception Specifications

Many of the operations on decimal floating-point types have wide contracts and, thus, cannot throw any exception. Where appropriate the corresponding operations should be declared to be `noexcept(true)`.

Note that the Decimal TR refers to “raising floating-point exceptions”. This doesn’t necessarily throw a C++ exception but may just setup an indication that a specific condition occurred. However, an implementation may choose to implement a mode of operation where C++ exceptions are thrown as a result of raising certain floating-point exceptions. Thus, the use of `noexcept(true)` probably won’t apply to many operations.

Constant Expressions

C++ 2011 added the ability to create `constexpr` functions. It may be desirable to turn certain operations into `constexpr` and it should be explicitly permitted to do so. In general, the operations shouldn’t be mandated to be `constexpr` because the semantics of many operations depend on run-time setting, e.g., because they use the rounding mode. On the other hand, the use of `constexpr` operations is especially desirable, e.g., because constant initialization is performed prior to any dynamic initialization (3.6.2, [basic.start.init], paragraph 2), thereby avoiding any issues relating to the order of initialization.

To really support the use of constant expressions for decimal floating-point types it is necessary to restrict the semantics of the operations. In particular, the operations need to be independent of the floating point environment (however, it seems ISO/IEC 60559 requires a way to specify the rounding mode to be used when computing constants). The conditional availability of the floating point environment would raise the requirement that functions can be overloaded on `constexpr` arguments, for example (in C++ 2011 it is not possible to overload these two functions):

```cpp
constexpr decimal64 operator+ (constexpr decimal64 d1,       
                             constexpr decimal64 d2);
decimal64          operator+ (decimal64 d1,               
                             decimal64 d2);
```

The first function would be used if the arguments `d1` and `d2` are constant expressions, otherwise the other function would be used. The implementations of the version using constant expressions wouldn’t raise any floating point exception and wouldn’t depend on the dynamically specified floating point context. The
implementations of both functions would do similar operations but possibly in vastly different ways. For example, the constant expression version would use a software implementation while the other version could be implemented to take advantage of hardware support for decimal floating points. However, corresponding support isn’t available in C++ 2011.

Not having `constexpr` support yields a viable library. If it is controversial to add `constexpr` it is probably safest to allow the use of `constexpr` but not to mandate it.

### Literal Suffixes

Section 4.1 (Literals) of the Decimal TR mentions that C uses literal suffixes for easy creation of decimal floating-point types. With the ability to define user-define literals a similar mechanism can be provided in C++. That is, the following operators should be added:

```cpp
template <char... C> constexpr decimal32 operator "" DF();
template <char... C> constexpr decimal64 operator "" DD();
template <char... C> constexpr decimal128 operator "" DL();
template <char... C> constexpr decimal32 operator "" df();
template <char... C> constexpr decimal64 operator "" dd();
template <char... C> constexpr decimal128 operator "" dl();
```

It may be desirable to mandate that the return types of these operators are `constexpr`. However, the implementations aren’t necessarily trivial. If mandated use of `constexpr` is controversial the support should only be allowed and not mandated.

### Decimal Formatting

Decimal floating points support a feature not available for binary floating points: They can represent the precision of the original number, i.e., they can keep track of trailing zeros after the decimal point (unless the number of digits would exceed the number of decimals for the decimal floating point). To support a choice of formatting the number using its own precision the C Decimal TR uses the `%a` and `%A` format specifiers which use the optionally present precision to restrict the formatted to number to a maximum number of digits.

Table 88 in 22.4.2.2.2 [facet.num.put.virtuals] paragraph 5 already specifies that the format specifiers `%a` and `%A` are used for floating point conversions when the `floatfield` is set to `std::ios_base::fixed | std::ios_base::scientific` (this is used to format binary floating point numbers using a hexadecimal format). The Decimal TR additionally expands the table for length modifiers to support the modifiers `H`, `D`, and `DD` for `decimal32`, `decimal64`, and `decimal128`, respectively. Thus, using `std::ios_base::fixed | std::ios_base::scientific` results in formatting decimal floating points taking their own precision into account when being formatted.

Unfortunately, this paragraph specifies that the precision (`str.precision()`) is only specified for floating point types if `floatfield != std::ios_base::fixed | std::ios_base::scientific`. However, it is desirable to optionally impose an upper bound on the used precision. One way to address this problem is to change the corresponding paragraph to become

```
For conversion from a floating type, if `floatfield != (ios_base::fixed |
ios_base::scientific)` or if a decimal floating-point type is formatted and 0 <
str.precision() is specified in the conversion specification. Otherwise, no precision is specified.

With this change the currently set precision would be taken into account when formatting decimal floating points. When str.precision() == 0 and floatfield is set to std::ios_base::fixed | std::ios_base::scientific no precision would be specified with the %a or %A specifiers when formatting a decimal floating point, i.e., its own precision is used. Note, that using a non-zero precision with the %a or %A format specifier affects all digits, not just the fractional digits (this is consistent with the way the %g and %G format specifiers work).

When setting floatfield to std::ios_base::fixed | std::ios_base::scientific it would be desirable that the default precision used is the decimal floating point’s own precision. This would imply that str.precision() == 0 and it seems unlikely that the default for str.precision() is changed. An alternative approach could be to use a new attribute on std::ios_base, e.g., decimal_precision(), which is used with formatting of decimal floating points and whose initial value is 0. To avoid any additional memory overhead this attribute could be accessed using the std::ios_base::iword().

Independent on how the precision is set, it may also be worth to add an alias for std::hexfloat which gives the operation a name meaningful in the context of decimal floating points. For example, std::decimal::ownprecision may be added which also sets the floatfield to std::ios_base::fixed | std::ios_base::scientific.

Concrete Changes to the Decimal TR ISO/IEC TR24733 (N2849)

This section describes the concrete changes to be made to the Decimal TR to produce the the updated revision of the Decimal TR.

Update References section:

Update the references section 1.3 to refer to an up to date revision of the International Standard for C++ and remove the reference to the TR on C++ Library Extensions:

1. In the first bullet (reference to the C++ standard), replace

   ISO/IEC 14882:2003

   by a reference to the the current revision of the C++ standard

   ISO/IEC 14883:2011

2. Since the additions made by the TR on C++ Library Extensions ISO/IEC TR 19768:2005 are incorporated into the International Standard for C++ remove the second bullet (reference to the library TR):

3. Add a reference to the C Technical Report on adding decimal floating-point arithmetic ISO/IEC TR 24732, i.e., add another bullet

ISO/IEC TR 24732, Information technology – Programming languages, their environments and system software interfaces – Extension for the programming language C to support decimal floating-point arithmetic

Remove the Relation to TR1

The relevant sections ([tr.meta], [tr.unord.fun.syn], [tr.unord.hash], and [tr.c99]) of the TR on C++ Library Extensions ISO/IEC TR 19768:2005 are incorporated into the International Standard for C++ Section 2.2 can be removed:

2.2 Relation to “Technical Report on C++ Library Extensions”

Unless otherwise specified, the following sections of ISO/IEC Technical Report 19768: Technical Report on C++ Library Extensions are included into this Technical Report by reference:

- General [tr.intro]
- Metaprogramming and type traits [tr.meta]
- Additions to header synopsis [tr.unord.fun.syn]
- Class template hash [tr.unord.hash]
- C-compatibility [tr.c99]

Remove Extensions to TR1 Headers Category

Section 2.3 (Categories of extensions) defines additions to TR1 changes as a separate category. Since the TR1 changes are incorporated into the International Standard for C++ these are identical to the one of the other categories (additions to existing standard headers or additions to standard library components). Thus, the third bullet can be removed from the list:

1. Change the first sentence to describe only 3 categories of library extension, i.e., replace

   This technical report describes 4 categories of library extensions:

   by

   This technical report describes 3 categories of library extensions:

2. Remove the third bullet of the list in Section 2.3:

   New library components declared as additions to TR1 headers, such as the template is_decimalFloating_point added to the header <type_traits> in subclause 3.11

3. Remove the reference to TR1 from the last sentence of the last paragraph, i.e., replace

   … are not present in the C++ standard or TR1.
by

... are not present in the C++ standard.

**Remove the Reference to Namespace std::tr1**

Section 2.4 describes reference to entities in the namespace std::tr1 and the assumption that these are qualified by std::tr1. With the entities being incorporated into the Internation Standard for C++ the names are now all in namespace std and the extra qualification can be removed.

Change the second paragraph of 2.4 from

> Unless otherwise specified, references to other entities described in this technical report are assumed to be qualified with std::decimal::, references to entities described in the C++ standard library are assumed to be qualified with std::, and references to entities described in TR1 are assumed to be qualified with std::tr1::.

... to become

> Unless otherwise specified, references to other entities described in this technical report are assumed to be qualified with std::decimal:: and references to entities described in the C++ standard library are assumed to be qualified with std::.

**Remove a Reference to a Change Category**

Since there are no changes to TR1 one components being made by the update, teh reference to “third” category needs to be removed from the second paragraph in Section 2.4. Change the first sentence of the third paragraph of Section 2.4 from

> Even when an extension is specified as additions to standard headers (the second and third categories in section 2.3), vendors should not simply add declarations to standard headers in a way that would be visible to users by default

... to become

> Even when an extension is specified as additions to standard headers (the second category in section 2.3), vendors should not simply add declarations to standard headers in a way that would be visible to users by default

**Require the Decimal Floating-Point Fytes to be Standard Layout Types**

As described above, C++11 generalized to the concept of PODs into the new category of standard layout types. The decimal floating points should be required to be standard layout types. In section 3 add a second paragraph:

> The decimal floating-point types are implemented as standard-layout types.
Add Decimal Literals to the Synopsis

In Section 3.2.1 add literal functions to the `<decimal>` synopsis, i.e., at the end of the header add

```cpp
// 3.2.12 Decimal Literals
template <char... C>
    constexpr decimal32 operator "" DF();
template <char... C>
    constexpr decimal64 operator "" DD();
template <char... C>
    constexpr decimal128 operator "" DL();
template <char... C>
    constexpr decimal32 operator "" df();
template <char... C>
    constexpr decimal64 operator "" dd();
template <char... C>
    constexpr decimal128 operator "" dl();
```

Make the Default Constructors Defaulted

Change the default constructors to be explicitly defaulted:

1. In Section 3.2.2 change the default constructor declaration from

   ```cpp
   decimal32();
   ```

   to become

   ```cpp
   decimal32() = default;
   ```

2. In Section 3.2.2.1 change the description of the default constructor from

   ```cpp
   decimal32();
   
   Effects: Constructs an object of type `decimal32` with the value equivalent to +0 and quantum equal to -101.
   ```

   to become

   ```cpp
   decimal32() = default;
   
   Effects: When used without any initialization the value of the constructed object is unspecified. When used in form causing zero initialization constructs an object of type `decimal32` with the value equivalent to +0 and quantum equal to -101.
   ```

3. In Section 3.2.3 change the default constructor declaration from

   ```cpp
   decimal64();
   ```

   to become

   ```cpp
   decimal64() = default;
   ```
4. In Section 3.2.3.1 change the description of the default constructor from

```
decimal64();
```

*Effects:* Constructs an object of type `decimal64` with the value equivalent to `+0` and quantum equal to `-398`.

to become

```
decimal64() = default;
```

*Effects:* When used without any initialization the value of the constructed object is unspecified. When used in form causing zero initialization constructs an object of type `decimal64` with the value equivalent to `+0` and quantum equal to `-398`.

5. In Section 3.2.4 change the default constructor declaration from

```
decimal128();
```

to become

```
decimal128() = default;
```

6. In Section 3.2.4.1 change the description of the default constructor from

```
decimal128();
```

*Effects:* Constructs an object of type `decimal128` with the value equivalent to `+0` and quantum equal to `-6176`.

to become

```
decimal128() = default;
```

*Effects:* When used without any initialization the value of the constructed object is unspecified. When used in form causing zero initialization constructs an object of type `decimal128` with the value equivalent to `+0` and quantum equal to `-6176`.

**Explicit Conversion Operators**

With the possibility of conversion operators being explicit, the existing conversions (i.e., to `long long`) should be made explicit and conversions to binary floating point types should be added:

1. In Section 3.2.2 change the declaration of the conversion operator from

   ```
   // conversion to integral types operator long long() const;
   ```

to become

   ```
   // conversions to built-in types explicit operator long long() const; explicit operator float()
   ```
const; explicit operator double() const; explicit operator long double() const;

2. In Section 3.2.2.4 change the description of the conversion operator from

**3.2.2.4 Conversion to integral type**

operator long long() const;

to become

**3.2.2.4 Conversion to built-in types**

explicit operator long long() const;

3. In Section 3.2.2.4 add descriptions of the conversion operators to binary floating point types, i.e., add

```cpp
explicit operator float() const;
```

*Returns:* If `std::numeric_limits<float>::is_iec559 == true`, returns the result of the conversion of `*this` to `float`, performed as in IEEE 754–2008. Otherwise, the returned value is implementation-defined.

```cpp
explicit operator double() const;
```

*Returns:* If `std::numeric_limits<double>::is_iec559 == true`, returns the result of the conversion of `*this` to `double`, performed as in IEEE 754–2008. Otherwise, the returned value is implementation-defined.

```cpp
explicit operator long double() const;
```

*Returns:* If `std::numeric_limits<long double>::is_iec559 == true`, returns the result of the conversion of `*this` to `long double`, performed as in IEEE 754–2008. Otherwise, the returned value is implementation-defined.

4. In Section 3.2.3 change the declaration of the conversion operator from

```cpp
// conversion to integral types operator long long() const;
```

to become

```cpp
// conversions to built-in types explicit operator long long() const; explicit operator float() const; explicit operator double() const; explicit operator long double() const;
```

5. In Section 3.2.3.4 change the description of the conversion operator from

**3.2.3.4 Conversion to integral type**

operator long long() const;

to become
3.2.3.4 Conversions to built-in types

explicit operator long long() const;

6. In Section 3.2.3.4 add descriptions of the conversion operators to binary floating point types, i.e., add

    explicit operator float() const;

*Returns:* If `std::numeric_limits<float>::is_iec559 == true`, returns the result of the conversion of `*this` to `float`, performed as in IEEE 754–2008. Otherwise, the returned value is implementation-defined.

explicit operator double() const;

*Returns:* If `std::numeric_limits<double>::is_iec559 == true`, returns the result of the conversion of `*this` to `double`, performed as in IEEE 754–2008. Otherwise, the returned value is implementation-defined.

explicit operator long double() const;

*Returns:* If `std::numeric_limits<long double>::is_iec559 == true`, returns the result of the conversion of `*this` to `long double`, performed as in IEEE 754–2008. Otherwise, the returned value is implementation-defined.

7. In Section 3.2.4 change the declaration of the conversion operator from

    // conversion to integral types operator long long() const;

to become

    // conversions to built-in types explicit operator long long() const; explicit operator float() const; explicit operator double() const; explicit operator long double() const;

8. In Section 3.2.4.4 change the description of the conversion operator from

    3.2.4.4 Conversion to integral type

    operator long long() const;

    to become

    3.2.4.4 Conversions to built-in types

    explicit operator long long() const;

9. In Section 3.2.3.4 add descriptions of the conversion operators to binary floating point types, i.e., add

    explicit operator float() const;

*Returns:* If `std::numeric_limits<float>::is_iec559 == true`, returns the result of the conversion of `*this` to `float`, performed as in IEEE 754–2008. Otherwise, the
returned value is implementation-defined.

explicit operator double() const;

>Returns: If std::numeric_limits<double>::is_iec559 == true, returns the result of the conversion of *this to double, performed as in IEEE 754–2008. Otherwise, the returned value is implementation-defined.

explicit operator long double() const;

>Returns: If std::numeric_limits<long double>::is_iec559 == true, returns the result of the conversion of *this to long double, performed as in IEEE 754–2008. Otherwise, the returned value is implementation-defined.

Make the Decimal Floating-Point types final

The floating-point types can’t be inherited from as they may be implemented like built-in types. Making them final prevents that. Thus, the class declarations should be made final:

1. In Section 3.2.2 change the declaration

   ```
   class decimal32 {
   }
   ```

   to become

   ```
   class decimal32 final {
   }
   ```

2. In Section 3.2.3 change the declaration

   ```
   class decimal64 {
   }
   ```

   to become

   ```
   class decimal64 final {
   }
   ```

3. In Section 3.2.4 change the declaration

   ```
   class decimal128 {
   }
   ```

   to become

   ```
   class decimal128 final {
   }
   ```

Remove Reference to C Literal Conversion

Secton 3.2.5 references Section 4.1 which describes that decimal literals are not supported. Since decimal literal support is being added, i.e., Section 4.1 becomes obsolete and references to it need to be removed. Remove the reference to Section 4.1 from the note at the end of the first paragraph of 3.2.5:
Also, see 4.1

Remove Reference to C Casts

Section 3.2.6 reference Section 4.2 which describes that there is no explicit conversion from decimal floating point types to binary floating points. With the availability of explicit conversion operations these are added, i.e., Section 4.2 becomes obsolete and references to it need to be removed:

1. Remove the reference to Section 4.2 from 3.2.6 paragraph 1:

   See 4.2

2. Remove the reference to Section 4.2 from 3.2.6 paragraph 2:

   See 4.2

3. Remove the reference to Section 4.2 from 3.2.6 paragraph 3:

   See 4.2

Add Description of the Literal Operators

The added literal operators need a specification. Add a new Section 3.2.12 defining the literal operators:

```cpp
template <char... C>
constexpr decimal32 operator °° DF();
template <char... C>
constexpr decimal64 operator °° DD();
template <char... C>
constexpr decimal128 operator °° DL();
template <char... C>
constexpr decimal32 operator °° df();
template <char... C>
constexpr decimal64 operator °° dd();
template <char... C>
constexpr decimal128 operator °° dl();
```

*Effects:* converts the string value to a decimal floating-point value as if parsing a `decimal32`, `decimal64`, or `decimal128` with `scanf()` using the conversion specifiers `%Hg`, `%Dg`, and `%DDg`, respectively. An invalid input string results in a compile-time error.

*Returns:* the result of the conversion.

Update Reference for the Description of `<cfloat>` and `<float.h>`

The headers `<cfloat>` and `<float.h>` are described in [c.limits]; Update the reference in 3.4, i.e., change the first sentence of 3.4 paragraph 1 from

The header `<cfloat>` is described in [tr.c99.cfloat]. The header `<float.h>` is described in [tr.c99.floath].
The headers `<cfloat>` and `<float.h>` are described in [c.limits].

**Update Reference for the Description of `<cfenv>` and `<fenv.h>`**

The headers `<cfenv>` and `<fenv.h>` are part of the International Standard for C++ and the reference to them needs to be updated. Change the first two sentences of 3.5 paragraph from

> The header `<cfenv>` is described in [tr.c99.cfenv]. The header `<fenv.h>` is described in [tr.c99.fenv].

to become

> The header `<cfenv>` is described in [cfenv]. The header `<fenv.h>` is described in [depr.c.headers].

**Remove TR1 Reference from Rounding Modes Table**

The Table 2 on DFP rounding direction macros references TR1. The corresponding macros are incorporated into the International Standard for C++. Change the table header for Table 2 from

> Equivalent TR1 macro for generic floating types

to become

> Equivalent macro for generic floating types

**Remove TR1 Reference in `<cmath>/math.h` Specification**

The comparison/classification functions are incorporated into the International Standard for C++ and the TR1 reference needs to be removed.

1. In Section 3.6 change the last sentence in paragraph 3 from

> The TR1 function templates `signbit, fpclassify, isinfinite, isinf, isnan, isnormal, isgreater, isgreateorequal, isless, islessequal, islessgreater, and isunordered` are also extended by this Technical Report to handle the decimal floating-point types.

   to become

> The comparison and classification function templates `signbit, fpclassify, isinfinite, isinf, isnan, isnormal, isgreater, isgreateorequal, isless, islessequal, islessgreater, and isunordered` are also extended by this Technical Report to handle the decimal floating-point types.

2. In Section 3.6.7 paragraph 1, change the first sentence from
For each of the following standard elementary functions from `<cmath>`, and for each of the following TR1 elementary functions from `<cmath>`:

to become

For each of the following standard elementary functions from `<cmath>`:

3. In Section 3.6.7 merge Tables 3 (functions originally from `<cmath>`) and 4 (functions originally introduced by TR1).

Remove TR1 reference for Type Traits

The type traits mentioned in Section 3.11 are now part the International Standard of C++ and live in namespace `std` instead of namespace `std::tr1`. Update the section to use the correct namespace, i.e. replace

```
std::tr1::is_arithmetic
std::tr1::is_fundamental
std::tr1::is_scalar
std::tr1::is_class
std::tr1::is_pod
```

However, the following expressions shall all yield the same Boolean value, where `dec` is one of `decimal32`, `decimal64`, or `decimal128`:

```
tr1::is_arithmetic<dec>::value
tr1::is_fundamental<dec>::value
tr1::is_scalar<dec>::value
!tr1::is_class<dec>::value
tr1::is_pod<dec>::value
```

[Note: The behavior of the type trait `std::tr1::is_floating_point` is not altered by this Technical Report. - end note]

by

The effect of the following type traits, when applied to any of the decimal floating-point types, is implementation-defined:

```
std::is_arithmetic
std::is_fundamental
std::is_scalar
std::is_class
std::is_pod
```

However, the following expressions shall all yield the same Boolean value, where `dec` is one of
decimal32, decimal64, or decimal128:

- `std::is_arithmetic<decimal>::value`
- `std::is_fundamental<decimal>::value`
- `std::is_scalar<decimal>::value`
- `!std::is_class<decimal>::value`
- `std::is_pod<decimal>::value`

[Note: The behavior of the type trait `std::is_floating_point` is not altered by this Technical Report. - end note]

Remove TR1 References from Hash Functions

The `hash` type is now incorporated into the International Standard for C++ and lives in namespace `std` instead of `std::tr1`. Update the namespace and the reference to the definition:

1. Change the declaration in 3.12.1 from

```cpp
namespace std {
namespace tr1 {
  // 3.12.2 Hash function specializations:
  template <> struct hash<decimal::decimal32>;
  template <> struct hash<decimal::decimal64>;
  template <> struct hash<decimal::decimal128>;
}
}
```

to become

```cpp
namespace std {
  // 3.12.2 Hash function specializations:
  template <> struct hash<decimal::decimal32>;
  template <> struct hash<decimal::decimal64>;
  template <> struct hash<decimal::decimal128>;
}
```

2. In 3.12.2 change the first sentence from

   In addition to the types indicated in [tr.unord.hash], the class template hash is required to be instantiable on the decimal floating-point types.

   to become

   In addition to the types indicated in [unord.hash], the class template hash is required to be instantiable on the decimal floating-point types.

Remove the Section on C Compatability

There were various incompatibilities to C pointed out in Section 4 which were addressed by new features added to C++ with the 2011 revision. As a result, the section isn’t needed and can be removed:
4 Notes on C compatibility

One of the goals of the design of the decimal floating-point types that are the subject of this Technical Report is to minimize incompatibility with the C decimal floating types; however, differences between the C and C++ languages make some incompatibility inevitable. Differences between the C and C++ decimal types—and techniques for overcoming them—are described in this section.

4.1 Literals

Literals of decimal floating-point type are not introduced to the C++ language by this Technical Report, though implementations may support them as a conforming extension. C programs that use decimal floating-point literals will not be portable to a C++ implementation that does not support this extension.

4.2 Conversions

In C, objects of decimal floating-point type can be converted to generic floating-point type by means of an explicit cast. In C++ this is not possible. Instead, the following functions should be used for this purpose:

- `decimal_to_float`
- `decimal_to_double`
- `decimal_to_long_double`
- `decimal32_to_float`
- `decimal32_to_double`
- `decimal32_to_long_double`
- `decimal64_to_float`
- `decimal64_to_double`
- `decimal64_to_long_double`
- `decimal128_to_float`
- `decimal128_to_double`
- `decimal128_to_long_double`

C programmers who wish to maintain portability to C++ should use these forms instead of the cast notation.

Revision History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2012–09–14</td>
<td>initial version</td>
</tr>
<tr>
<td>2</td>
<td>2014–01–19</td>
<td>- updated the dates for expected standard revisions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- added the concrete changes to apply to the decimal TR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- some editorial corrections</td>
</tr>
</tbody>
</table>