ISO/IEC JTC1 SC22 WG21 N2732

Date: 2008-08-22

Reference number of document: **ISO/IEC WDTR 24733**

Committee identification: ISO/IEC JTC1 SC22 WG21

SC22 Secretariat: ANSI

Information Technology —

Programming languages, their environments and system software interfaces —

Extension for the programming language C++ to support decimal floating-point arithmetic —

Warning

This document is an ISO/IEC draft Technical Report. It is not an ISO/IEC International Technical Report. It is distributed for review and comment. It is subject to change without notice and shall not be referred to as an International Technical Report or International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: Technical Report Type 2
Document subtype: n/a
Document stage: (2) Working Draft Technical Report
Document language: E

Copyright notice

This ISO document is a working draft or committee draft and is copyright-protected by ISO.

Requests for permission to reproduce this document for the purpose of selling it should be addressed as shown below or to ISO's member body in the country of the requester:

ISO copyright office
Case postale 56
CH-1211 Geneva 20
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Reproduction for sales purposes may be subject to royalty payments or a licensing agreement.

Violators may be prosecuted.

1 Introduction

Most of today's general purpose computing architectures provide binary floating-point arithmetic in hardware. Binary float-point is an efficient representation that minimizes memory use, and is simpler to implement than floating-point arithmetic using other bases. It has therefore become the norm for scientific computations, with almost all implementations following the IEEE-754 standard for binary floating-point arithmetic.

However, human computation and communication of numeric values almost always uses decimal arithmetic, and decimal notations. Laboratory notes, scientific papers, legal documents, business reports and financial statements all record numeric values in decimal form. When numeric data are given to a program or are displayed to a user, binary to-and-from decimal conversion is required. There are inherent rounding errors involved in such conversions; decimal fractions cannot, in general, be represented exactly by floating-point values. These errors often cause usability and efficiency problems, depending on the application.

These problems are minor when the application domain accepts, or requires results to have, associated error estimates (as is the case with scientific applications). However, in business and financial applications, computations are either required to be exact (with no rounding errors) unless explicitly rounded, or be supported by detailed analyses that are auditable to be correct. Such applications therefore have to take special care in handling any rounding errors introduced by the computations.

The most efficient way to avoid conversion error is to use decimal arithmetic. Recognizing this, the IEEE 754-2008~~R~~ Standard for Floating-Point Arithmetic specifies decimal floating-point encodings and arithmetic. This technical report specifies extensions to the International Standard for the C++ programming language to permit the use of decimal arithmetic in a manner consistent with the IEEE 754-2008~~R~~ standard.

1.1 Arithmetic model

This Technical Report is based on a model of decimal arithmetic 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 IEEE 754-2008~~R~~.

There are three components to the model:

- *~~numbers~~ data* - ~~which represent the values~~ numbers and NaNs, which can be manipulated by, or be the results of, the core operations defined in the model

- *operations* - ~~the core operations~~ (such as addition, multiplication, etc.) which can be carried out on ~~numbers~~ data

- *context* – ~~which represents the user-selectable parameters and rules which govern the results of arithmetic operations (for example, the rounding mode to be used)~~ the status of operations (namely, exceptions flags), and controls to govern the results of operations (for example, rounding modes).

The model defines these components in the abstract. It defines neither the way in which operations are expressed (which might vary depending on the computer language or other interface being used), nor the concrete representation (specific layout in storage, or in a processor's register, for example) of ~~numbers~~ data or context.

From the perspective of the C++ language, *~~numbers~~ data* are represented by data types, *operations* are defined within expressions, and *context* is the floating environment specified in `<cfenv>` and `<fenv.h>`. This Technical Report specifies how the C++ language implements these components.

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

1.2 ~~Encodings~~ The Formats

In the C++ International Standard, the representation of a floating-point number is specified in an abstract form where the constituent components of the representation are defined (sign, exponent, significand) but the internals of these components are not. 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.

This approach was inherited from the C programming language. At the time that C was first standardized, there were already various hardware implementations of floating-point arithmetic in common use. Specifying the exact details of a representation would make most of the existing C implementations at the time not conforming.

The C99 standard specifies a binding to IEEE-754 (annex F). Still, conformance to IEEE-754 is not a mandatory requirement. A C99 implementation that conforms to IEEE-754 defines the macro __STDC_IEC_559__.

This Technical Report specifies decimal floating-point arithmetic according to the IEEE 754-2008~~R~~ standard, with the constituent components of the representation defined. This is a more stringent than the approach taken for the floating point types in the C++ standard. Since it is expected that all decimal floating-point hardware implementations will conform to the IEEE 754-2008~~R~~ standard, binding to this standard directly benefits both implementers and programmers.

1.3 References

The following standards contain provisions which, through reference in this text, constitute provisions of this Technical Report. For dated references, subsequent amendment to, or revisions of, any of these publications do not apply. However, parties to agreements based on this Technical Report are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred applies. Members of the IEC and ISO maintain registers of current valid International Standards.

1.3.1 ISO/IEC 14882:2003*, Information technology -- Programming languages, their environments and system software interfaces -- Programming Language C++*.

1.3.2 ISO/IEC TR 19768:2005, *Information technology -- Programming languages, their environments and system software interfaces -- Technical Report on C++ Library Extensions*.

1.3.3 ANSI/IEEE 754-2008~~R~~ - *IEEE Standard for Floating-Point Arithmetic. The Institute of Electrical and Electronic Engineers, Inc.*.

1.3.4 ANSI/IEEE 854-1997 - *IEEE Standard for Radix-Independent Floating-Point Arithmetic. The Institute of Electrical and Electronic Engineers, Inc., New York*, 1987.

2 Conventions

This technical report is non-normative; the extensions that it describes may be considered for inclusion in a future revision of the International Standard for C++, but they are not currently required for conformance to that standard. Furthermore, it is conceivable that a future revision of the International Standard will include facilities that are similar and not identical to the extensions described in this report,

Although this report describes extensions to the *C++ standard library*, vendors may choose to implement these extensions in the C++ language translator itself. This practice is permitted so long as all well-formed programs are accepted by the implementation, and the semantics of those programs are the same as they would be had the extensions taken the form of a library. [*Note:* The same practice is permitted with respect to the implementation of the C++ standard library. --*end note*]

The result of deriving a user-defined type from `std::decimal::decimal32`, `std::decimal::decimal64`, or `std::decimal::decimal128` is undefined.

2.1 Relation to C++ Standard Library Introduction

Unless otherwise specified, the whole of the ISO C++ Standard Library introduction [lib.library] is included into this Technical Report by reference.

This Technical Report introduces the following elements to supplement those described in [lib.structure.specifications]:

Expansion: the semantics of a macro's expansion text

Library Overload Set: an overload set for a given function name or overloaded operator. Each library overload set specifies the parameter types for the required overloads, and can be implemented as a suitably constrained function template, a set of overloaded non-template functions, or a combination of both.

[*Example:*

Consider a library overload set with the following specification:

```
decimal128 foo(decimal128 d, RHS rhs);
```

> **Library Overload Set:** *RHS* may be any of the decimal floating-point types..
> **Returns:** *d*.

A possible implementation of this library overload set is:

```
decimal128 foo(decimal128 d, decimal128) { return d; }
decimal128 foo(decimal128 d, decimal64)  { return d; }
decimal128 foo(decimal128 d, decimal32)  { return d; }
```

Another possible implementation of this library overload set is:

```
template <typename RHS>
decimal128 foo(decimal128 d, RHS) { return d; }
```

In the latter implementation, the template type parameter *RHS* must be constrained to the decimal floating-point types.

--*end example*]

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 `<functional>` synopsis [tr.unord.fun.syn]
- Class template `hash` [tr.unord.hash]
- C compatibility [tr.c99]

2.3 Categories of extensions

This technical report describes 4 categories of library extensions:

1. New library components (types and functions) that are declared entirely in new headers, such as the type `decimal::decimal32` in the `<decimal>` header.

2. New library components declared as additions to existing standard headers, such as the functions added to the headers `<cmath>` and `<math.h>` in subclause 3.6.

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

4. Additions to standard library components, such as the specializations of `std::numeric_limits` in subclause 3.3.

New headers are distinguished from extensions to existing headers by the title of the *synopsis* clause. In the first case the title is of the form "Header `<foo>` synopsis," and the synopsis includes all namespace scope declarations contained in the header. In the second case the title is of the form "Additions to header `<foo>` synopsis" and the synopsis includes only the extensions, *i.e.* those namespace scope declarations that are not present in the C++ standard or TR1.

2.4 Namespaces and headers

The extensions described in this technical report are declared within the namespace `decimal`, which is nested inside the namespace `std`.

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::`.

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 [*Note:* That would fail to be standard conforming, because the new names, even within a namespace, could conflict with user macros. --*end note*] Users should be required to take explicit action to have access to library extensions. It is recommended either that additional declarations in standard headers be protected with a macro that is not defined by default, or else that all extended headers, including both new headers and parallel versions of standard headers with nonstandard declarations, be placed in a separate directory that is not part of the default search path.

3 Decimal floating-point types

This Technical Report introduces three *decimal floating-point types*, named decimal32, decimal64, and decimal128. The set of values of type decimal32 is a subset of the set of values of type decimal64; the set of values of the type decimal64 is a subset of the set of values of the type decimal128. ~~Support for decimal128 is optional.~~ These types supplement the standard C++ types `float`, `double`, and `long double`, which are collectively described as the *basic floating types*.

3.1 ~~Decimal type encodings~~ Characteristics of decimal floating-point types

The three decimal encoding formats defined in IEEE 754-2008 correspond to the three decimal floating types as follows:

- decimal32 is a *decimal32* number, which is encoded in four consecutive octets (32 bits)
- decimal64 is a *decimal64* number, which is encoded in eight consecutive octets (64 bits)
- decimal128 is a *decimal128* number, which is encoded in 16 consecutive octets (128 bits)

~~The finite numbers are defined by a sign, an exponent, (which is a power of ten), and a decimal integer coefficient.~~ The value of a finite number is given by $(-1)^{\text{sign}}$ x coefficient x 10^{exponent}. Refer to IEEE 754-2008 for details of the format.

These formats are characterized by the length of the coefficient, and the maximum and minimum exponent. The coefficient is not normalized, so trailing zeros are significant; i.e. 1.0 is equal to but can be distinguished from 1.00. Table 1 shows these characteristics by format.

Table 1 -- Format Characteristics

Format	**decimal32**	**decimal64**	**decimal128**
Coefficient length in digits	7	16	34
Maximum Exponent (E_{max})	96	385	6145
Minimum Exponent (E_{min})	-95	-382	-6142

3.2 Decimal Types

3.2.1 Header `<decimal>` synopsis

```
#include <iosfwd>

namespace std {
namespace decimal {

  // 3.2.2 class decimal32:
  class decimal32;

  // 3.2.3 class decimal64:
  class decimal64;

  // 3.2.4 class decimal128:
  class decimal128;

  // 3.2.5 initialization from coefficient and exponent:
  decimal32  make_decimal32 (long long coeff, int exponent);
  decimal32  make_decimal32 (unsigned long long coeff,
                             int exponent);
  decimal64  make_decimal64 (long long coeff, int exponent);
  decimal64  make_decimal64 (unsigned long long coeff,
                             int exponent);
  decimal128 make_decimal128(long long coeff, int exponent);
  decimal128 make_decimal128(unsigned long long coeff,
                             int exponent);

  // 3.2.6 conversion to generic floating-point type:
  long double decimal32_to_long_double (decimal32 d);
  long double decimal64_to_long_double (decimal64 d);
  long double decimal128_to_long_double(decimal128 d);
  long double decimal_to_long_double(decimal32 d);
  long double decimal_to_long_double(decimal64 d);
  long double decimal_to_long_double(decimal128 d);

  // 3.2.7 unary arithmetic operators:
  decimal32  operator+(decimal32  rhs);
  decimal64  operator+(decimal64  rhs);
  decimal128 operator+(decimal128 rhs);
  decimal32  operator-(decimal32  rhs);
  decimal64  operator-(decimal64  rhs);
  decimal128 operator-(decimal128 rhs);

  // 3.2.8 binary arithmetic operators:
  /* see 3.2.8 */ operator+(LHS lhs, decimal32  rhs);
  /* see 3.2.8 */ operator+(LHS lhs, decimal64  rhs);
  decimal128      operator+(LHS lhs, decimal128 rhs);

  decimal32  operator+(decimal32  lhs, RHS rhs);
  decimal64  operator+(decimal64  lhs, RHS rhs);
  decimal128 operator+(decimal128 lhs, RHS rhs);

  /* see 3.2.8 */ operator-(LHS lhs, decimal32  rhs);
  /* see 3.2.8 */ operator-(LHS lhs, decimal64  rhs);
```

```
decimal128      operator-(LHS lhs, decimal128 rhs);

decimal32  operator-(decimal32  lhs, RHS rhs);
decimal64  operator-(decimal64  lhs, RHS rhs);
decimal128 operator-(decimal128 lhs, RHS rhs);

/* see 3.2.8 */ operator*(LHS lhs, decimal32  rhs);
/* see 3.2.8 */ operator*(LHS lhs, decimal64  rhs);
decimal128      operator*(LHS lhs, decimal128 rhs);

decimal32  operator*(decimal32  lhs, RHS rhs);
decimal64  operator*(decimal64  lhs, RHS rhs);
decimal128 operator*(decimal128 lhs, RHS rhs);

/* see 3.2.8 */ operator/(LHS lhs, decimal32  rhs);
/* see 3.2.8 */ operator/(LHS lhs, decimal64  rhs);
decimal128      operator/(LHS lhs, decimal128 rhs);

decimal32  operator/(decimal32  lhs, RHS rhs);
decimal64  operator/(decimal64  lhs, RHS rhs);
decimal128 operator/(decimal128 lhs, RHS rhs);

// 3.2.9 comparison operators:
bool operator==(LHS lhs, decimal32  rhs);
bool operator==(LHS lhs, decimal64  rhs);
bool operator==(LHS lhs, decimal128 rhs);

bool operator==(decimal32  lhs, RHS rhs);
bool operator==(decimal64  lhs, RHS rhs);
bool operator==(decimal128 lhs, RHS rhs);

bool operator!=(LHS lhs, decimal32  rhs);
bool operator!=(LHS lhs, decimal64  rhs);
bool operator!=(LHS lhs, decimal128 rhs);

bool operator!=(decimal32  lhs, RHS rhs);
bool operator!=(decimal64  lhs, RHS rhs);
bool operator!=(decimal128 lhs, RHS rhs);

bool operator<(LHS lhs, decimal32  rhs);
bool operator<(LHS lhs, decimal64  rhs);
bool operator<(LHS lhs, decimal128 rhs);

bool operator<(decimal32  lhs, RHS rhs);
bool operator<(decimal64  lhs, RHS rhs);
bool operator<(decimal128 lhs, RHS rhs);

bool operator<=(LHS lhs, decimal32  rhs);
bool operator<=(LHS lhs, decimal64  rhs);
bool operator<=(LHS lhs, decimal128 rhs);

bool operator<=(decimal32  lhs, RHS rhs);
bool operator<=(decimal64  lhs, RHS rhs);
bool operator<=(decimal128 lhs, RHS rhs);

bool operator>(LHS lhs, decimal32  rhs);
bool operator>(LHS lhs, decimal64  rhs);
```

```
  bool operator>(LHS lhs, decimal128 rhs);

  bool operator>(decimal32  lhs, RHS rhs);
  bool operator>(decimal64  lhs, RHS rhs);
  bool operator>(decimal128 lhs, RHS rhs);

  bool operator>=(LHS lhs, decimal32  rhs);
  bool operator>=(LHS lhs, decimal64  rhs);
  bool operator>=(LHS lhs, decimal128 rhs);

  bool operator>=(decimal32  lhs, RHS rhs);
  bool operator>=(decimal64  lhs, RHS rhs);
  bool operator>=(decimal128 lhs, RHS rhs);

  // 3.2.10 Formatted input:
  template <class charT, class traits>
    std::basic_istream<charT, traits> &
      operator>>(std::basic_istream<charT, traits> & is,
                 decimal32 & d);

  template <class charT, class traits>
    std::basic_istream<charT, traits> &
      operator>>(std::basic_istream<charT, traits> & is,
                 decimal64 & d);

  template <class charT, class traits>
    std::basic_istream<charT, traits> &
      operator>>(std::basic_istream<charT, traits> & is,
                 decimal128 & d);

  // 3.2.11 Formatted output:
  template <class charT, class traits>
    std::basic_ostream<charT, traits> &
      operator<<(std::basic_ostream<charT, traits> & os,
                 decimal32 d);

  template <class charT, class traits>
    std::basic_ostream<charT, traits> &
      operator<<(std::basic_ostream<charT, traits> & os,
                 decimal64 d);

  template <class charT, class traits>
    std::basic_ostream<charT, traits> &
      operator<<(std::basic_ostream<charT, traits> & os,
                 decimal128 d);
}
}
```

3.2.2 Class `decimal32`

```
namespace std {
namespace decimal {
  class decimal32 {
    public:
      // 3.2.2.1 construct/copy/destroy:
      decimal32();

      // 3.2.2.2 conversion from floating-point type:
      explicit decimal32(decimal64 d64);
      explicit decimal32(decimal128 d128);
      explicit decimal32(float r);
      explicit decimal32(double r);
      explicit decimal32(long double r);

      // 3.2.2.3 conversion from integral type:
      decimal32(int z);
      decimal32(unsigned int z);
      decimal32(long z);
      decimal32(unsigned long z);
      decimal32(long long z);
      decimal32(unsigned long long z);

      // 3.2.2.4 conversion to integral type:
      operator long long() const;

      // 3.2.2.5 increment and decrement operators:
      decimal32 & operator++();
      decimal32   operator++(int);
      decimal32 & operator--();
      decimal32   operator--(int);

      // 3.2.2.6 compound assignment:
      decimal32 & operator+=(RHS rhs);
      decimal32 & operator-=(RHS rhs);
      decimal32 & operator*=(RHS rhs);
      decimal32 & operator/=(RHS rhs);
  };
}
}
```

3.2.2.1 Construct/copy/destroy

```
decimal32();
```

Effects: Constructs an object of type decimal32 with the value equivalent to 0 and quantum equal to `DEC32_MINEXP`.

3.2.2.2 Conversion from floating-point type

```
explicit decimal32(decimal64 d64);
```

Effects: Constructs an object of type decimal32 by converting from type decimal64. Conversion is performed as in IEEE 754-2008.

```
explicit decimal32(decimal128 d128);
```

Effects: Constructs an object of type decimal32 by converting from type decimal128.

Conversion is performed as in IEEE 754-2008.

```
        explicit decimal32(float r);
```

Effects: Constructs an object of type decimal32 by converting from type `float`. If `std::numeric_limits<float>::is_iec559 == true` then the conversion is performed as in IEEE 754-2008. Otherwise, the result of the conversion is implementation-defined.

```
        explicit decimal32(double r);
```

Effects: Constructs an object of type decimal32 by converting from type `double`. If `std::numeric_limits<double>::is_iec559 == true` then the conversion is performed as in IEEE 754-2008. Otherwise, the result of the conversion is implementation-defined.

```
        explicit decimal32(long double r);
```

Effects: Constructs an object of type decimal32 by converting from type `long double`. If `std::numeric_limits<long double>::is_iec559 == true` then the conversion is performed as in IEEE 754-2008. Otherwise, the result of the conversion is implementation-defined.

3.2.2.3 Conversion from integral type

```
        decimal32(int z);
        decimal32(unsigned int z);
        decimal32(long z);
        decimal32(unsigned long z);
        decimal32(long long z);
        decimal32(unsigned long long z);
```

Effects: Constructs an object of type decimal32 by converting from the type of *z*. Conversion is performed as in IEEE 754-2008. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result shall be correctly rounded with exceptions raised as in IEEE 754-2008.

3.2.2.4 Conversion to integral type

```
        operator long long() const;
```

Returns: Returns the result of the conversion of `*this` to the type `long long`, as if performed by the expression `llroundd32(*this)` while the decimal rounding direction mode [3.5.2] `FE_DEC_TOWARD_ZERO` is in effect.

3.2.2.5 Increment and decrement operators

```
        decimal32 & operator++();
```

Effects: Adds 1 to `*this`, as in IEEE 754-2008, and assigns the result to `*this`.
Returns: `*this`

```
        decimal32   operator++(int);
```

Effects:

```
        decimal32 tmp = *this;
        *this += 1;
        return tmp;
```

```
decimal32 & operator--();
```

Effects: Subtracts 1 from `*this`, as in IEEE 754-2008, and assigns the result to `*this`. **Returns:** `*this`

```
decimal32   operator--(int);
```

Effects:

```
decimal32 tmp = *this;
*this -= 1;
return tmp;
```

3.2.2.6 Compound assignment

```
decimal32 & operator+=(RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types, or any of the decimal floating-point types. **Effects:** Adds *rhs* to `*this`, as in IEEE 754-2008, and assigns the result to `*this`. **Returns:** `*this`

```
decimal32 & operator-=(RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types, or any of the decimal floating-point types. **Effects:** Subtracts *rhs* from `*this`, as in IEEE 754-2008, and assigns the result to `*this`. **Returns:** `*this`

```
decimal32 & operator*=(RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types, or any of the decimal floating-point types. **Effects:** Multiplies `*this` by *rhs*, as in IEEE 754-2008, and assigns the result to `*this`. **Returns:** `*this`

```
decimal32 & operator/=(RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types, or any of the decimal floating-point types. **Effects:** Divides `*this` by *rhs*, as in IEEE 754-2008, and assigns the result to `*this`. **Returns:** `*this`

3.2.3 Class `decimal64`

```
namespace std {
namespace decimal {
  class decimal64 {
    public:
      // 3.2.3.1 construct/copy/destroy:
      decimal64();

      // 3.2.3.2 conversion from floating-point type:
               decimal64(decimal32 d32);
      explicit decimal64(decimal128 d128);
      explicit decimal64(float r);
      explicit decimal64(double r);
      explicit decimal64(long double r);

      // 3.2.3.3 conversion from integral type:
      decimal64(int z);
      decimal64(unsigned int z);
```

```
        decimal64(long z);
        decimal64(unsigned long z);
        decimal64(long long z);
        decimal64(unsigned long long z);

        // 3.2.3.4 conversion to integral type:
        operator long long() const;

        // 3.2.3.5 increment and decrement operators:
        decimal64 & operator++();
        decimal64   operator++(int);
        decimal64 & operator--();
        decimal64   operator--(int);

        // 3.2.3.6 compound assignment:
        decimal64 & operator+=(RHS rhs);
        decimal64 & operator-=(RHS rhs);
        decimal64 & operator*=(RHS rhs);
        decimal64 & operator/=(RHS rhs);
    };
  }
  }
```

3.2.3.1 Construct/copy/destroy

```
decimal64();
```

Effects: Constructs an object of type decimal64 with the value equivalent to 0 and quantum equal to `DEC64_MINEXP`.

3.2.3.2 Conversion from floating-point type

```
decimal64(decimal32 d32);
```

Effects: Constructs an object of type decimal64 by converting from type decimal32. Conversion is performed as in IEEE 754-2008.

```
explicit decimal64(decimal128 d128);
```

Effects: Constructs an object of type decimal64 by converting from type decimal128. Conversion is performed as in IEEE 754-2008.

```
explicit decimal64(float r);
```

Effects: Constructs an object of type decimal64 by converting from type `float`. If `std::numeric_limits<float>::is_iec559 == true` then the conversion is performed as in IEEE 754-2008. Otherwise, the result of the conversion is implementation-defined.

```
explicit decimal64(double r);
```

Effects: Constructs an object of type decimal64 by converting from type `double`. If `std::numeric_limits<double>::is_iec559 == true` then the conversion is performed as in IEEE 754-2008. Otherwise, the result of the conversion is implementation-defined.

```
explicit decimal64(long double r);
```

Effects: Constructs an object of type decimal64 by converting from type `long double`. If `std::numeric_limits<long double>::is_iec559 == true` then the conversion is performed as in IEEE 754-2008. Otherwise, the result of the conversion is implementation-defined.

3.2.3.3 Conversion from integral type

```
decimal64(int z);
decimal64(unsigned int z);
decimal64(long z);
decimal64(unsigned long z);
decimal64(long long z);
decimal64(unsigned long long z);
```

Effects: Constructs an object of type decimal64 by converting from the type of *z*. Conversion is performed as in IEEE 754-2008. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result shall be correctly rounded with exceptions raised as in IEEE 754-2008.

3.2.3.4 Conversion to integral type

```
operator long long() const;
```

Returns: Returns the result of the conversion of `*this` to the type `long long`, as if performed by the expression `llroundd64(*this)` while the decimal rounding direction mode [3.5.2] `FE_DEC_TOWARD_ZERO` is in effect. .

3.2.3.5 Increment and decrement operators

```
decimal64 & operator++();
```

Effects: Adds 1 to `*this`, as in IEEE 754-2008, and assigns the result to `*this`.
Returns: `*this`

```
decimal64   operator++(int);
```

Effects:

```
Decimal64 tmp = *this;
*this += 1;
return tmp;
```

```
decimal64 & operator--();
```

Effects: Subtracts 1 from `*this`, as in IEEE 754-2008, and assigns the result to `*this`.
Returns: `*this`

```
decimal64   operator--(int);
```

Effects:

```
decimal64 tmp = *this;
*this -= 1;
return tmp;
```

3.2.3.6 Compound assignment

```
decimal64 & operator+=(RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types, or any of the decimal floating-point types. **Effects:** Adds *rhs* to `*this`, as in IEEE 754-2008, and assigns the result to `*this`. **Returns:** `*this`

```
decimal64 & operator-=(RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types, or any of the decimal floating-point types. **Effects:** Subtracts *rhs* from `*this`, as in IEEE 754-2008, and assigns the result to `*this`. **Returns:** `*this`

```
decimal64 & operator*=(RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types, or any of the decimal floating-point types. **Effects:** Multiplies `*this` by *rhs*, as in IEEE 754-2008, and assigns the result to `*this`. **Returns:** `*this`

```
decimal64 & operator/=(RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types, or any of the decimal floating-point types. **Effects:** Divides `*this` by *rhs*, as in IEEE 754-2008, and assigns the result to `*this`. **Returns:** `*this`

3.2.4 Class `decimal128`

```
namespace std {
namespace decimal {
  class decimal128 {
    public:
      // 3.2.4.1 construct/copy/destroy:
      decimal128();

      // 3.2.4.2 conversion from floating-point type:
               decimal128(decimal32 d32);
               decimal128(decimal64 d64);
      explicit decimal128(float r);
      explicit decimal128(double r);
      explicit decimal128(long double r);

      // 3.2.4.3 conversion from integral type:
      decimal128(int z);
      decimal128(unsigned int z);
      decimal128(long z);
      decimal128(unsigned long z);
      decimal128(long long z);
      decimal128(unsigned long long z);

      // 3.2.4.4 conversion to integral type:
      operator long long() const;

      // 3.2.4.5 increment and decrement operators:
      decimal128 & operator++();
      decimal128   operator++(int);
      decimal128 & operator--();
      decimal128   operator--(int);
```

```
            // 3.2.4.6 compound assignment:
            decimal128 & operator+=(RHS rhs);
            decimal128 & operator-=(RHS rhs);
            decimal128 & operator*=(RHS rhs);
            decimal128 & operator/=(RHS rhs);
        };
      }
      }
```

3.2.4.1 Construct/copy/destroy

```
decimal128();
```

Effects: Constructs an object of type decimal128 with the value equivalent to 0 and quantum equal to `DEC128_MINEXP`.

3.2.4.2 Conversion from floating-point type

```
decimal128(decimal32 d32);
```

Effects: Constructs an object of type decimal128 by converting from type decimal32. Conversion is performed as in IEEE 754-2008.

```
decimal128(decimal64 d64);
```

Effects: Constructs an object of type decimal128 by converting from type decimal64. Conversion is performed as in IEEE 754-2008.

```
explicit decimal128(float r);
```

Effects: Constructs an object of type decimal128 by converting from type `float`. If `std::numeric_limits<float>::is_iec559 == true` then the conversion is performed as in IEEE 754-2008. Otherwise, the result of the conversion is implementation-defined.

```
explicit decimal128(double r);
```

Effects: Constructs an object of type decimal128 by converting from type `double`. If `std::numeric_limits<double>::is_iec559 == true` then the conversion is performed as in IEEE 754-2008. Otherwise, the result of the conversion is implementation-defined.

```
explicit decimal128(long double r);
```

Effects: Constructs an object of type decimal128 by converting from type `long double`. If `std::numeric_limits<long double>::is_iec559 == true` then the conversion is performed as in IEEE 754-2008. Otherwise, the result of the conversion is implementation-defined.

3.2.4.3 Conversion from integral type

```
decimal128(int z);
decimal128(unsigned int z);
decimal128(long z);
decimal128(unsigned long z);
decimal128(long long z);
decimal128(unsigned long long z);
```

Effects: Constructs an object of type decimal128 by converting from the type of *z*. Conversion is performed as in IEEE 754-2008. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result shall be correctly rounded with exceptions raised as in IEEE 754-2008.

3.2.4.4 Conversion to integral type

```
        operator long long() const;
```

Returns: Returns the result of the conversion of `*this` to the type `long long`, as if performed by the expression `llroundd128(*this)` while the decimal rounding direction mode [3.5.2] `FE_DEC_TOWARD_ZERO` is in effect. .

3.2.4.5 Increment and decrement operators

```
        decimal128 & operator++();
```

Effects: Adds 1 to `*this`, as in IEEE 754-2008, and assigns the result to `*this`.
Returns: `*this`

```
        decimal128   operator++(int);
```

Effects:

```
        Decimal128 tmp = *this;
        *this += 1;
        return tmp;

        decimal128 & operator--();
```

Effects: Subtracts 1 from `*this`, as in IEEE 754-2008, and assigns the result to `*this`.
Returns: `*this`

```
  decimal128   operator--(int);
```

Effects:

```
        decimal128 tmp = *this;
        *this -= 1;
        return tmp;
```

3.2.4.6 Compound assignment

```
        decimal128 & operator+=(RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types, or any of the decimal floating-point types. **Effects:** Adds *rhs* to `*this`, as in IEEE 754-2008, and assigns the result to `*this`. **Returns:** `*this`

```
        decimal128 & operator-=(RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types, or any of the decimal floating-point types. **Effects:** Subtracts *rhs* from `*this`, as in IEEE 754-2008, and assigns the result to `*this`. **Returns:** `*this`

```
        decimal128 & operator*=(RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types, or any of the decimal floating-point types. **Effects:** Multiplies `*this` by *rhs*, as in IEEE 754-2008, and assigns the result to `*this`. **Returns:** `*this`

```
        decimal128 & operator/=(RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types, or any of the decimal floating-point types. **Effects:** Divides `*this` by *rhs*, as in IEEE 754-2008, and assigns the result to `*this`. **Returns:** `*this`

3.2.5 Initialization from coefficient and exponent

```
decimal32  make_decimal32 (long long coeff, int exponent);
decimal32  make_decimal32 (unsigned long long coeff,
                           int exponent);
decimal64  make_decimal64 (long long coeff, int exponent);
decimal64  make_decimal64 (unsigned long long coeff,
                           int exponent);
decimal128 make_decimal128(long long coeff, int exponent);
decimal128 make_decimal128(unsigned long long coeff,
                           int exponent);
```

Effects: If the value *coeff* x $10^{exponent}$ is outside of the range of values that can be represented by the return type, plus or minus HUGE_VAL_D32, HUGE_VAL_D64, or HUGE_VAL_D128 is returned (according to the return type and the sign of *coeff*), and the value of the macro ERANGE is stored in errno. If the result underflows, plus or minus DEC32_MIN, DEC64_MIN, or DEC128_MIN is returned (according to the return type and the sign of *coeff*), and the value of ERANGE is stored in errno. Otherwise, returns an object of the appropriate decimal floating-point type with the value *coeff* x $10^{exponent}$.

3.2.6 Conversion to generic floating-point type

```
long double decimal32_to_long_double (decimal32 d);
long double decimal64_to_long_double (decimal64 d);
long double decimal128_to_long_double(decimal128 d);
long double decimal_to_long_double(decimal32  d);
long double decimal_to_long_double(decimal64  d);
long double decimal_to_long_double(decimal128 d);
```

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

3.2.7 Unary arithmetic operators

```
decimal32  operator+(decimal32  lhs);
decimal64  operator+(decimal64  lhs);
decimal128 operator+(decimal128 lhs);
```

Returns: Adds *lhs* to `0`, as in IEEE 754-2008, and returns the result.

```
decimal32  operator-(decimal32  lhs);
decimal64  operator-(decimal64  lhs);
decimal128 operator-(decimal128 lhs);
```

Returns: ~~Subtracts *lhs* from `0`, as in IEEE 754-2008, and~~ returns the result of inverting the sign of *lhs*.

[*Note:* because of the possibility that *lhs* may equal -0, this is not the same as subtracting *lhs* from 0 –*end note*]

3.2.8 Binary arithmetic operators

The return type of each binary arithmetic operator in this section is determined according to the following algorithm:

- if one or both of the parameters of the overloaded operator is decimal128, then the return type is decimal128,
- otherwise, if one or both of the parameters is decimal64, then the return type is decimal64,
- otherwise, the return type is decimal32.

```
/* see above */ operator+(LHS lhs, decimal32  rhs);
/* see above */ operator+(LHS lhs, decimal64  rhs);
decimal128      operator+(LHS lhs, decimal128 rhs);
```

Library Overload Set: *LHS* may be any of the integral types or any of the decimal floating-point types. **Returns:** Adds *rhs* to *lhs*, as in IEEE 754-2008, and returns the result.

```
decimal32  operator+(decimal32  lhs, RHS rhs);
decimal64  operator+(decimal64  lhs, RHS rhs);
decimal128 operator+(decimal128 lhs, RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types. **Returns:** Adds *rhs* to *lhs*, as in IEEE 754-2008, and returns the result.

```
/* see above */ operator-(LHS lhs, decimal32  rhs);
/* see above */ operator-(LHS lhs, decimal64  rhs);
decimal128      operator-(LHS lhs, decimal128 rhs);
```

Library Overload Set: *LHS* may be any of the integral types or any of the decimal floating-point types. **Returns:** Subtracts *rhs* from *lhs*, as in IEEE 754-2008, and returns the result.

```
decimal32  operator-(decimal32  lhs, RHS rhs);
decimal64  operator-(decimal64  lhs, RHS rhs);
decimal128 operator-(decimal128 lhs, RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types. **Returns:** Subtracts *rhs* from *lhs*, as in IEEE 754-2008, and returns the result.

```
/* see above */ operator*(LHS lhs, decimal32  rhs);
/* see above */ operator*(LHS lhs, decimal64  rhs);
decimal128      operator*(LHS lhs, decimal128 rhs);
```

Library Overload Set: *LHS* may be any of the integral types or any of the decimal floating-point types. **Returns:** Multiplies *lhs* by *rhs*, as in IEEE 754-2008, and returns the result.

```
decimal32  operator*(decimal32  lhs, RHS rhs);
decimal64  operator*(decimal64  lhs, RHS rhs);
decimal128 operator*(decimal128 lhs, RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types. **Returns:** Multiplies *lhs* by *rhs*, as in IEEE 754-2008, and returns the result.

```
/* see above */ operator/(LHS lhs, decimal32  rhs);
/* see above */ operator/(LHS lhs, decimal64  rhs);
decimal128      operator/(LHS lhs, decimal128 rhs);
```

Library Overload Set: *LHS* may be any of the integral types or any of the decimal floating-point types. **Returns:** Divides *lhs* by *rhs*, as in IEEE 754-2008, and returns the result.

```
decimal32  operator/(decimal32  lhs, RHS rhs);
decimal64  operator/(decimal64  lhs, RHS rhs);
decimal128 operator/(decimal128 lhs, RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types. **Returns:** Divides *lhs* by *rhs*, as in IEEE 754-2008, and returns the result.

3.2.9 Comparison operators

```
bool operator==(LHS lhs, decimal32  rhs);
bool operator==(LHS lhs, decimal64  rhs);
bool operator==(LHS lhs, decimal128 rhs);
```

Library Overload Set: *LHS* may be any of the integral types or any of the decimal floating-point types. **Returns:** `true` if *lhs* is exactly equal to *rhs* according to IEEE 754-2008, `false` otherwise.

```
bool operator==(decimal32  lhs, RHS rhs);
bool operator==(decimal64  lhs, RHS rhs);
bool operator==(decimal128 lhs, RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types. **Returns:** `true` if *lhs* is exactly equal to *rhs* according to IEEE 754-2008, `false` otherwise.

```
bool operator!=(LHS lhs, decimal32  rhs);
bool operator!=(LHS lhs, decimal64  rhs);
bool operator!=(LHS lhs, decimal128 rhs);
```

Library Overload Set: *LHS* may be any of the integral types or any of the decimal floating-point types. **Returns:** `true` if *lhs* is not exactly equal to *rhs* according to IEEE 754-2008, `false` otherwise.

```
bool operator!=(decimal32  lhs, RHS rhs);
bool operator!=(decimal64  lhs, RHS rhs);
bool operator!=(decimal128 lhs, RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types. **Returns:** `true` if *lhs* is not exactly equal to *rhs* according to IEEE 754-2008, `false` otherwise.

```
bool operator<(LHS lhs, decimal32  rhs);
bool operator<(LHS lhs, decimal64  rhs);
bool operator<(LHS lhs, decimal128 rhs);
```

Library Overload Set: *LHS* may be any of the integral types or any of the decimal floating-point types. **Returns:** `true` if *lhs* is less than *rhs* according to IEEE 754-2008, `false` otherwise.

```
bool operator<(decimal32  lhs, RHS rhs);
bool operator<(decimal64  lhs, RHS rhs);
bool operator<(decimal128 lhs, RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types. **Returns:** `true` if *lhs* is less than *rhs* according to IEEE 754-2008, `false` otherwise.

```
bool operator<=(LHS lhs, decimal32  rhs);
bool operator<=(LHS lhs, decimal64  rhs);
bool operator<=(LHS lhs, decimal128 rhs);
```

Library Overload Set: *LHS* may be any of the integral types or any of the decimal floating-point types. **Returns:** `true` if *lhs* is less than or equal to *rhs* according to IEEE 754-2008, `false` otherwise.

```
bool operator<=(decimal32  lhs, RHS rhs);
bool operator<=(decimal64  lhs, RHS rhs);
bool operator<=(decimal128 lhs, RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types. **Returns:** `true` if *lhs* is less than or equal to *rhs* according to IEEE 754-2008, `false` otherwise.

```
bool operator>(LHS lhs, decimal32  rhs);
bool operator>(LHS lhs, decimal64  rhs);
bool operator>(LHS lhs, decimal128 rhs);
```

Library Overload Set: *LHS* may be any of the integral types or any of the decimal floating-point types. **Returns:** `true` if *lhs* is greater than *rhs* according to IEEE 754-2008, `false` otherwise.

```
bool operator>(decimal32  lhs, RHS rhs);
bool operator>(decimal64  lhs, RHS rhs);
bool operator>(decimal128 lhs, RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types. **Returns:** `true` if *lhs* is greater than *rhs* according to IEEE 754-2008, `false` otherwise.

```
bool operator>=(LHS lhs, decimal32  rhs);
bool operator>=(LHS lhs, decimal64  rhs);
bool operator>=(LHS lhs, decimal128 rhs);
```

Library Overload Set: *LHS* may be any of the integral types or any of the decimal floating-point types. **Returns:** `true` if *lhs* is greater than or equal to *rhs* according to IEEE 754-2008, `false` otherwise.

```
bool operator>=(decimal32  lhs, RHS rhs);
bool operator>=(decimal64  lhs, RHS rhs);
bool operator>=(decimal128 lhs, RHS rhs);
```

Library Overload Set: *RHS* may be any of the integral types. **Returns:** `true` if *lhs* is greater than or equal to *rhs* according to IEEE 754-2008, `false` otherwise.

3.2.10 Formatted input

```
template <class charT, class traits>
  std::basic_istream<charT, traits> &
    operator>>(std::basic_istream<charT, traits> & is,
               decimal32  & d);

template <class charT, class traits>
  std::basic_istream<charT, traits> &
    operator>>(std::basic_istream<charT, traits> & is,
               decimal64  & d);

template <class charT, class traits>
```

```
          std::basic_istream<charT, traits> &
            operator>>(std::basic_istream<charT, traits> & is,
                       decimal128 & d);
```

Effects: This function constructs an object of class `std::basic_istream<charT, traits>::sentry`. If the `sentry` object returns `true` when converted to a value of type bool, input is extracted as if by the following code fragment:

```
typedef extended_num_get<charT,
       std::istreambuf_iterator<charT, traits> > extnumget;
std::ios_base::iostate err = 0;
std::use_facet<extnumget>(is.getloc()).get(*this, 0, *this,
                                           err, d);
setstate(err);
```

If an exception is thrown during input then `std::ios::badbit` is set in the error state of the input stream *is*. If `(is.exceptions() & std::ios_base::badbit) != 0` then the exception is rethrown. In any case, the formatted input function destroys the `sentry` object.

Returns: *is*.

3.2.11 Formatted output

```
        template <class charT, class traits>
          std::basic_ostream<charT, traits> &
            operator<<(std::basic_ostream<charT, traits> & os,
                       decimal32  d);

        template <class charT, class traits>
          std::basic_ostream<charT, traits> &
            operator<<(std::basic_ostream<charT, traits> & os,
                       decimal64  d);

        template <class charT, class traits>
          std::basic_ostream<charT, traits> &
            operator<<(std::basic_ostream<charT, traits> & os,
                       decimal128 d);
```

Effects: This function constructs an object of class `std::basic_ostream<charT, traits>::sentry`. If the `sentry` object returns `true` when converted to a value of type bool, output is generated as if by the following code fragment:

```
typedef extended_num_put<charT,
    std::ostreambuf_iterator<charT, traits> > extnumput;
bool failed =
    std::use_facet<extnumput>(os.getloc()).put(*this,
        *this, os.fill(), d).failed();
if (failed)
  { os.setstate(std::ios_base::failbit); }
```

If an exception is thrown during output then `std::ios::badbit` is set in the error state of the input stream *os*. If `(os.exceptions() & std::ios_base::badbit) != 0` then the exception is rethrown. In any case, the formatted output function destroys the `sentry` object.

Returns: *os*.

3.3 Additions to header `<limits>`

The standard template `std::numeric_limits` shall be specialized for the `decimal32`, `decimal64`, and `decimal128` types.

[This space intentionally blank.] [*Example:* `namespace std {`
`template<> class numeric_limits<decimal::decimal32> {` `public:`
`static const bool is_specialized = true;`

```
            static decimal::decimal32 min() throw()
                { return DEC32_MIN; }
            static decimal::decimal32 max() throw()
                { return DEC32_MAX; }

            static const int digits       = 7;
            static const int digits10     = digits;
            static const int max_digits10 = digits;

            static const bool is_signed  = true;
            static const bool is_integer = false;
            static const bool is_exact   = false;

            static const int radix = 10;
            static decimal::decimal32 epsilon()     throw()
                { return DEC32_EPSILON; }
            static decimal::decimal32 round_error() throw()
                { return ...; }

            static const int min_exponent   = -95;
            static const int min_exponent10 = min_exponent;
            static const int max_exponent   = 96;
            static const int max_exponent10 = max_exponent;

            static const bool has_infinity             = true;
            static const bool has_quiet_NaN            = true;
            static const bool has_signaling_NaN        = true;
            static const float_denorm_style has_denorm =
                denorm_present;
            static const bool has_denorm_loss          = true;

            static decimal::decimal32 infinity()      throw()
                { return ...; }
            static decimal::decimal32 quiet_NaN()     throw()
                { return ...; }
            static decimal::decimal32 signaling_NaN() throw()
                { return ...; }
            static decimal::decimal32 denorm_min()    throw()
                { return DEC32_DEN; }

            static const bool is_iec559       = false;
            static const bool is_bounded      = true;
            static const bool is_modulo       = false;
            static const bool traps           = true;
            static const bool tinyness_before = true;

            static const float_round_style round_style =
                round_indeterminate;
        };

        template<> class numeric_limits<decimal::decimal64> {
          public:
            static const bool is_specialized = true;

            static decimal::decimal64 min() throw()
                { return DEC64_MIN; }
            static decimal::decimal64 max() throw()
                { return DEC64_MAX; }

            static const int digits       = 16;
            static const int digits10     = digits;
            static const int max_digits10 = digits;

            static const bool is_signed   = true;
            static const bool is_integer  = false;
            static const bool is_exact    = false;

            static const int radix = 10;
            static decimal::decimal64 epsilon()     throw()
                { return DEC64_EPSILON; }
            static decimal::decimal64 round_error() throw()
                { return ...; }

            static const int min_exponent   = -382;
            static const int min_exponent10 = min_exponent;
            static const int max_exponent   = 385;
            static const int max_exponent10 = max_exponent;

            static const bool has_infinity             = true;
            static const bool has_quiet_NaN            = true;
            static const bool has_signaling_NaN        = true;
            static const float_denorm_style has_denorm =
                denorm_present;
            static const bool has_denorm_loss          = true;

            static decimal::decimal64 infinity()      throw()
                { return ...; }
            static decimal::decimal64 quiet_NaN()     throw()
                { return ...; }
            static decimal::decimal64 signaling_NaN() throw()
                { return ...; }
            static decimal::decimal64 denorm_min()    throw()
                { return DEC64_DEN; }

            static const bool is_iec559       = false;
            static const bool is_bounded      = true;
            static const bool is_modulo       = false;
            static const bool traps           = true;
            static const bool tinyness_before = true;

            static const float_round_style round_style =
                round_indeterminate;
        };
```

```
  template<> class numeric_limits<decimal::decimal128> {
    public:
      static const bool is_specialized = true;

      static decimal::decimal128 min() throw()
          { return DEC128_MIN; }
      static decimal::decimal128 max() throw()
          { return DEC128_MAX; }

      static const int digits       = 34;
      static const int digits10     = digits;
      static const int max_digits10 = digits;

      static const bool is_signed  = true;
      static const bool is_integer = false;
      static const bool is_exact   = false;

      static const int radix = 10;
      static decimal::decimal128 epsilon()     throw()
          { return DEC128_EPSILON; }
      static decimal::decimal128 round_error() throw()
          { return ...; }

      static const int min_exponent   = -6142;
      static const int min_exponent10 = min_exponent;
      static const int max_exponent   = 6145;
      static const int max_exponent10 = max_exponent;

      static const bool has_infinity             = true;
      static const bool has_quiet_NaN            = true;
      static const bool has_signaling_NaN        = true;
      static const float_denorm_style has_denorm =
          denorm_present;
      static const bool has_denorm_loss          = true;

      static decimal::decimal128 infinity()      throw()
          { return ...; }
      static decimal::decimal128 quiet_NaN()     throw()
          { return ...; }
      static decimal::decimal128 signaling_NaN() throw()
          { return ...; }
      static decimal::decimal128 denorm_min()    throw()
          { return DEC128_DEN; }

      static const bool is_iec559       = false;
      static const bool is_bounded      = true;
      static const bool is_modulo       = false;
      static const bool traps           = true;
      static const bool tinyness_before = true;

      static const float_round_style round_style =
          round_indeterminate;
  };
}  --end example]
```

3.4 Headers `<cfloat>` and `<float.h>`

The header `<cdecfloat>` is described in [tr.c99.cfloat]. The header `<float.h>` is described in [tr.c99.floath]. These headers are extended by this Technical Report to define characteristics of the decimal floating-point types `decimal32`, `decimal64`, and `decimal128`. As well, `<float.h>` is extended to define the convenience typedefs `_Decimal32`, `_Decimal64`, and `_Decimal128` for compatibilty with the C programming language.

3.4.1 Header `<cfloat>` synopsis

```
#include <decimal>

// number of digits in the coefficient:
#define DEC32_MANT_DIG  7
#define DEC64_MANT_DIG 16
#define DEC64_MANT_DIG 34

// minimum exponent:
#define DEC32_MIN_EXP    -95
#define DEC64_MIN_EXP    ~~-383~~ -382
#define DEC128_MIN_EXP -6143

// maximum exponent:
#define ~~DEC32_MIN_EXP~~ DEC32_MAX_EXP    96
#define ~~DEC64_MIN_EXP~~ DEC64_MAX_EXP   ~~384~~ 385
#define ~~DEC128_MIN_EXP~~ DEC128_MAX_EXP 6144

// 3.4.3 maximum finite value:
#define DEC32_MAX      implementation-defined
#define DEC64_MAX      implementation-defined
#define DEC128_MAX     implementation-defined

// 3.4.4 epsilon:
#define DEC32_EPSILON   implementation-defined
#define DEC64_EPSILON   implementation-defined
#define DEC128_EPSILON  implementation-defined

// 3.4.5 minimum positive normal value:
#define DEC32_MIN      implementation-defined
#define DEC64_MIN      implementation-defined
#define DEC128_MIN     implementation-defined

// 3.4.6 minimum positive subnormal value:
#define ~~DEC32_DEN~~ DEC32_SUBNORMAL implementation-defined
#define ~~DEC64_DEN~~ DEC64_SUBNORMAL implementation-defined
#define ~~DEC128_DEN~~ DEC128_SUBNORMAL implementation-defined

// 3.4.7 evaluation format:
#define DEC_EVAL_METHOD implementation-defined
```

3.4.2 Header `<float.h>` synopsis

```
// C-compatibility convenience typedefs:
typedef std::decimal::decimal32  _Decimal32;
typedef std::decimal::decimal64  _Decimal64;
typedef std::decimal::decimal128 _Decimal128;
```

3.4.3 Maximum finite value

```
#define DEC32_MAX  implementation-defined
```

Expansion: an rvalue of type `decimal32` equal to the maximum finite number that can be represented by an object of type `decimal32`; exactly equal to 9.999999 x 10^{96} (there are six 9's after the decimal point)

```
#define DEC64_MAX  implementation-defined
```

Expansion: an rvalue of type `decimal64` equal to the maximum finite number that can be represented by an object of type `decimal64`; exactly equal to 9.999999999999999 x 10^{384} (there are fifteen 9's after the decimal point)

```
#define DEC128_MAX implementation-defined
```

Expansion: an rvalue of type `decimal128` equal to the maximum finite number that can be represented by an object of type `decimal128`; exactly equal to 9.999999999999999999999999999999999 x 10^{6144} (there are thirty-three 9's after the decimal point)

3.4.4 Epsilon

```
#define DEC32_EPSILON  implementation-defined
```

Expansion: an rvalue of type `decimal32` equal to the difference between 1 and the least value greater than 1 that can be represented by an object of type `decimal32`; exactly equal to 1 x 10^{-6}

```
#define DEC64_EPSILON  implementation-defined
```

Expansion: an rvalue of type `decimal64` equal to the difference between 1 and the least value greater than 1 that can be represented by an object of type `decimal64`; exactly equal to 1 x 10^{-15}

```
#define DEC128_EPSILON implementation-defined
```

Expansion: an rvalue of type `decimal128` equal to the difference between 1 and the least value greater than 1 that can be represented by an object of type `decimal128`; exactly equal to 1 x 10^{-33}

3.4.5 Minimum positive normal value

```
#define DEC32_MIN  implementation-defined
```

Expansion: an rvalue of type `decimal32` equal to the minimum positive normal number that can be represented by an object of type `decimal32`; exactly equal to 1 x 10^{-95}

```
#define DEC64_MIN  implementation-defined
```

Expansion: an rvalue of type `decimal64` equal to the minimum positive normal number that can be represented by an object of type `decimal64`; exactly equal to 1 x 10^{-383}

```
#define DEC128_MIN implementation-defined
```

Expansion: an rvalue of type `decimal128` equal to the minimum positive normal number that can be represented by an object of type `decimal128`; exactly equal to 1 x 10^{-6143}

3.4.6 Minimum positive subnormal value

#define ~~DEC32_DEN~~ DEC32_SUBNORMAL **implementation-defined**

Expansion: an rvalue of type `decimal32` equal to the minimum positive finite number that can be represented by an object of type `decimal32`; exactly equal to 1 x 10^{-95}

#define ~~DEC64_DEN~~ DEC64_SUBNORMAL **implementation-defined**

Expansion: an rvalue of type `decimal64` equal to the minimum positive finite number that can be represented by an object of type `decimal64`; exactly equal to 1 x 10^{-383}

#define ~~DEC128_DEN~~ DEC128_SUBNORMAL **implementation-defined**

Expansion: an rvalue of type `decimal128` equal to the minimum positive finite number that can be represented by an object of type `decimal128`; exactly equal to
1 x 10^{-6143}

3.4.7 Evaluation format

```
#define DEC_EVAL_METHOD implementation-defined
```

Except for assignment and casts, the values of operations with decimal floating operands and values subject to the usual arithmetic conversions 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 `DEC_EVAL_METHOD`:

```
-1 indeterminable;

 0 evaluate all operations and constants just to the range
   and precision of the type;

 1 evaluate operations and constants of type decimal32 and
   decimal64 to the range and precision of the decimal64 type,
   evaluate decimal128 operations and constants to the range
   and precision of the decimal128 type;

 2 evaluate all operations and constants to the range and
   precision of the decimal128 type.
```

All other negative values for `DEC_EVAL_METHOD` characterize implementation-defined behavior.

3.5 Additions to `<cfenv>` and `<fenv.h>`

The header `<cfenv>` is described in [tr.c99.cfenv]. The header `<fenv.h>` is described in [tr.c99.fenv]. The floating point environment specified in these subclauses is extended by this Technical Report to apply to decimal floating-point types.

3.5.1 Additions to `<cfenv>` synopsis

```
// 3.5.2 rounding direction macros:
#define FE_DEC_DOWNWARD            implementation-defined
#define FE_DEC_TONEAREST           implementation-defined
#define FE_DEC_TONEARESTFROMZERO implementation-defined
#define FE_DEC_TOWARD_ZERO         implementation-defined
#define FE_DEC_UPWARD              implementation-defined

namespace std {
namespace decimal {

  // 3.5.3 fe_dec_getround function:
  int fe_dec_getround();

  // 3.5.4 fe_dec_setround function:
  int fe_dec_setround(int round);
}
}
```

3.5.2 Rounding modes

Macros are added to `<cfenv>` and `<fenv.h>`:

Table 2 -- DFP rounding direction macros

Additional DFP rounding direction macros introduced by this Technical Report	**Equivalent TR1 macro for generic floating types**	**IEEE-754**
FE_DEC_DOWNWARD	FE_DOWNWARD	Towards minus infinity
FE_DEC_TONEAREST	FE_TONEAREST	To nearest, ties even
FE_DEC_TONEARESTFROMZERO	n/a	To nearest, ties away from zero
FE_DEC_TOWARD_ZERO	FE_TOWARD_ZERO	Toward zero

FE_DEC_UPWARD	FE_UPWARD	Toward plus infinity

These macros are used by the `fegetround` and `fesetround` functions for getting and setting the rounding mode to be used in decimal floating-point operations.

3.5.3 The `fe_dec_getround` function

```
int fe_dec_getround();
```

Effects: gets the current rounding direction for decimal floating-point operations.

Returns: the value of the rounding direction macro representing the current rounding direction for decimal floating-point operations, or a negative value if there is no such rounding macro or the current rounding direction is not determinable.

3.5.4 The `fe_dec_setround` function

```
int fe_dec_setround(int round);
```

Effects: establishes *round* as the rounding direction for decimal floating-point operations. If *round* is not equal to the value of a DFP rounding direction macro, the rounding direction is not changed.

If `FLT_RADIX` is not 10, the rounding direction altered by the `fesetround` function is independent of the rounding direction altered by the `fe_dec_setround` function; otherwise, if `FLT_RADIX` is 10, whether the `fesetround` and `fe_dec_setround` functions alter the rounding direction of both generic floating type and decimal floating type operations is implementation defined.

Returns: a zero value if and only if the argument is equal to one of the rounding direction macros introduced in 3.6.2.

3.5.5 Changes to `<fenv.h>`

Each name placed into the namespace `decimal` by `<cfenv>` is placed into both the namespace `decimal` and the global namespace by `<fenv.h>`.

3.6 Additions to `<cmath>` and `<math.h>`

The elementary mathematical functions declared in the standard C++ header `<cmath>` are overloaded by this Technical Report to support the decimal floating-point types. The macros `HUGE_VAL_D32`, `HUGE_VAL_D64`, `HUGE_VAL_D128`, `DEC_INFINITY`, and `DEC_NAN` are defined for use with these functions. With the exception of `sqrt`, `fmax`, and `fmin`, the accuracy of the result of a call to one of these functions is implementation-defined. The implementation may state that the accuracy is unknown. The TR1 function templates `signbit`, `fpclassify`, `isinfinite`, `isinf`, `isnan`, `isnormal`, `isgreater`, `isgreaterequal`, `isless`, `islessequal`, `islessgreater`, and `isunordered` are also extended by this Technical Report to handle the decimal floating-point types.

3.6.1 Additions to header `<cmath>` synopsis

```
// 3.6.2 macros:
#define HUGE_VAL_D32    implementation-defined
#define HUGE_VAL_D64    implementation-defined
#define HUGE_VAL_D128   implementation-defined
#define DEC_INFINITY    implementation-defined
#define DEC_NAN         implementation-defined
#define FP_FAST_FMAD32  implementation-defined
#define FP_FAST_FMAD64  implementation-defined
#define FP_FAST_FMAD128 implementation-defined

namespace std {
namespace decimal {

  // 3.6.3 evaluation formats:
  typedef decimal-floating-type decimal32_t;
  typedef decimal-floating-type decimal64_t;

  // 3.6.4 samequantum functions:
  bool samequantum     (decimal32 x,  decimal32 y);
  bool samequantumd32  (decimal32 x,  decimal32 y);

  bool samequantum     (decimal64 x,  decimal64 y);
  bool samequantumd64  (decimal64 x,  decimal64 y);

  bool samequantum     (decimal128 x, decimal128 y);
  bool samequantumd128 (decimal128 x, decimal128 y);

  // 3.6.5 quantize functions:
  decimal32  quantize     (decimal32 x,  decimal32 y);
  decimal32  quantized32  (decimal32 x,  decimal32 y);

  decimal64  quantize     (decimal64 x,  decimal64 y);
  decimal64  quantized64  (decimal64 x,  decimal64 y);

  decimal128 quantize     (decimal128 x, decimal128 y);
  decimal128 quantized128 (decimal128 x, decimal128 y);

  // 3.6.6 elementary functions:
  // trigonometric functions:
  decimal32  acosd32  (decimal32  x);
  decimal64  acosd64  (decimal64  x);
  decimal128 acosd128 (decimal128 x);

  decimal32  asind32  (decimal32  x);
  decimal64  asind64  (decimal64  x);
  decimal128 asind128 (decimal128 x);

  decimal32  atand32  (decimal32  x);
  decimal64  atand64  (decimal64  x);
  decimal128 atand128 (decimal128 x);

  decimal32  atan2d32  (decimal32  x, decimal32  y);
  decimal64  atan2d64  (decimal64  x, decimal64  y);
```

```
decimal128 atan2d128 (decimal128 x, decimal128 y);

decimal32  cosd32  (decimal32  x);
decimal64  cosd64  (decimal64  x);
decimal128 cosd128 (decimal128 x);

decimal32  sind32  (decimal32  x);
decimal64  sind64  (decimal64  x);
decimal128 sind128 (decimal128 x);

decimal32  tand32  (decimal32  x);
decimal64  tand64  (decimal64  x);
decimal128 tand128 (decimal128 x);

// hyperbolic functions:
decimal32  acoshd32  (decimal32  x);
decimal64  acoshd64  (decimal64  x);
decimal128 acoshd128 (decimal128 x);

decimal32  asinhd32  (decimal32  x);
decimal64  asinhd64  (decimal64  x);
decimal128 asinhd128 (decimal128 x);

decimal32  atanhd32  (decimal32  x);
decimal64  atanhd64  (decimal64  x);
decimal128 atanhd128 (decimal128 x);

decimal32  coshd32  (decimal32  x);
decimal64  coshd64  (decimal64  x);
decimal128 coshd128 (decimal128 x);

decimal32  sinhd32  (decimal32  x);
decimal64  sinhd64  (decimal64  x);
decimal128 sinhd128 (decimal128 x);

decimal32  tanhd32  (decimal32  x);
decimal64  tanhd64  (decimal64  x);
decimal128 tanhd128 (decimal128 x);

// exponential and logarithmic functions:
decimal32  expd32  (decimal32  x);
decimal64  expd64  (decimal64  x);
decimal128 expd128 (decimal128 x);

decimal32  exp2d32  (decimal32  x);
decimal64  exp2d64  (decimal64  x);
decimal128 exp2d128 (decimal128 x);

decimal32  expm1d32  (decimal32  x);
decimal64  expm1d64  (decimal64  x);
decimal128 expm1d128 (decimal128 x);

decimal32  frexpd32  (decimal32  value, int * exp);
decimal64  frexpd64  (decimal64  value, int * exp);
decimal128 frexpd128 (decimal128 value, int * exp);

int ilogbd32  (decimal32  x);
int ilogbd64  (decimal64  x);
int ilogbd128 (decimal128 x);

decimal32  ldexpd32  (decimal32  x, int exp);
decimal64  ldexpd64  (decimal64  x, int exp);
decimal128 ldexpd128 (decimal128 x, int exp);

decimal32  logd32  (decimal32  x);
decimal64  logd64  (decimal64  x);
decimal128 logd128 (decimal128 x);

decimal32  log10d32  (decimal32  x);
decimal64  log10d64  (decimal64  x);
decimal128 log10d128 (decimal128 x);

decimal32  log1pd32  (decimal32  x);
decimal64  log1pd64  (decimal64  x);
decimal128 log1pd128 (decimal128 x);

decimal32  log2d32  (decimal32  x);
decimal64  log2d64  (decimal64  x);
decimal128 log2d128 (decimal128 x);

decimal32  logbd32  (decimal32  x);
decimal64  logbd64  (decimal64  x);
decimal128 logbd128 (decimal128 x);

decimal32  modfd32  (decimal32  value, decimal32  * iptr);
decimal64  modfd64  (decimal64  value, decimal64  * iptr);
decimal32  modfd128 (decimal128 value, decimal128 * iptr);

decimal32  scalbnd32  (decimal32  x, int n);
decimal64  scalbnd64  (decimal64  x, int n);
decimal128 scalbnd128 (decimal128 x, int n);

decimal32  scalblnd32  (decimal32  x, long int n);
decimal64  scalblnd64  (decimal64  x, long int n);
decimal128 scalblnd128 (decimal128 x, long int n);

// power and absolute-value functions:
decimal32  cbrtd32  (decimal32  x);
decimal64  cbrtd64  (decimal64  x);
decimal128 cbrtd128 (decimal128 x);

decimal32  fabsd32  (decimal32  x);
decimal64  fabsd64  (decimal64  x);
decimal128 fabsd128 (decimal128 x);

decimal32  hypotd32  (decimal32  x, decimal32  y);
decimal64  hypotd64  (decimal64  x, decimal64  y);
decimal128 hypotd128 (decimal128 x, decimal128 y);

decimal32  powd32  (decimal32  x, decimal32  y);
decimal64  powd64  (decimal64  x, decimal64  y);
decimal128 powd128 (decimal128 x, decimal128 y);

decimal32  sqrtd32  (decimal32  x);
```

```
decimal64  sqrtd64  (decimal64  x);
decimal128 sqrtd128 (decimal128 x);

// error and gamma functions:
decimal32  erfd32  (decimal32  x);
decimal64  erfd64  (decimal64  x);
decimal128 erfd128 (decimal128 x);

decimal32  erfcd32  (decimal32  x);
decimal64  erfcd64  (decimal64  x);
decimal128 erfcd128 (decimal128 x);

decimal32  lgammad32  (decimal32  x);
decimal64  lgammad64  (decimal64  x);
decimal128 lgammad128 (decimal128 x);

decimal32  tgammad32  (decimal32  x);
decimal64  tgammad64  (decimal64  x);
decimal128 tgammad128 (decimal128 x);

// nearest integer functions:
decimal32  ceild32  (decimal32  x);
decimal64  ceild64  (decimal64  x);
decimal128 ceild128 (decimal128 x);

decimal32  floord32  (decimal32  x);
decimal64  floord64  (decimal64  x);
decimal128 floord128 (decimal128 x);

decimal32  nearbyintd32  (decimal32  x);
decimal64  nearbyintd64  (decimal64  x);
decimal128 nearbyintd128 (decimal128 x);

decimal32  rintd32  (decimal32  x);
decimal64  rintd64  (decimal64  x);
decimal128 rintd128 (decimal128 x);

long int lrintd32  (decimal32  x);
long int lrintd64  (decimal64  x);
long int lrintd128 (decimal128 x);

long long int llrintd32  (decimal32  x);
long long int llrintd64  (decimal64  x);
long long int llrintd128 (decimal128 x);

decimal32  roundd32  (decimal32  x);
decimal64  roundd64  (decimal64  x);
decimal128 roundd128 (decimal128 x);

long int lroundd32  (decimal32  x);
long int lroundd64  (decimal64  x);
long int lroundd128 (decimal128 x);

long long int llroundd32  (decimal32  x);
long long int llroundd64  (decimal64  x);
long long int llroundd128 (decimal128 x);
```

```
  decimal32  truncd32  (decimal32  x);
  decimal64  truncd64  (decimal64  x);
  decimal128 truncd128 (decimal128 x);

  // remainder functions:
  decimal32  fmodd32  (decimal32  x, decimal32  y);
  decimal64  fmodd64  (decimal64  x, decimal64  y);
  decimal128 fmodd128 (decimal128 x, decimal128 y);

  decimal32  remainderd32  (decimal32  x, decimal32  y);
  decimal64  remainderd64  (decimal64  x, decimal64  y);
  decimal128 remainderd128 (decimal128 x, decimal128 y);

  decimal32  remquod32  (decimal32  x, decimal32  y, int * quo);
  decimal64  remquod64  (decimal64  x, decimal64  y, int * quo);
  decimal128 remquod128 (decimal128 x, decimal128 y, int * quo);

  // manipulation functions:
  decimal32  copysignd32  (decimal32  x, decimal32  y);
  decimal64  copysignd64  (decimal64  x, decimal64  y);
  decimal128 copysignd128 (decimal128 x, decimal128 y);

  decimal32  nand32  (const char * tagp);
  decimal64  nand64  (const char * tagp);
  decimal128 nand128 (const char * tagp);

  decimal32  nextafterd32  (decimal32  x, decimal32  y);
  decimal64  nextafterd64  (decimal64  x, decimal64  y);
  decimal128 nextafterd128 (decimal128 x, decimal128 y);

  decimal32  nexttowardd32  (decimal32  x, decimal32  y);
  decimal64  nexttowardd64  (decimal64  x, decimal64  y);
  decimal128 nexttowardd128 (decimal128 x, decimal128 y);

  // maximum, minimum, and positive difference functions:
  decimal32  fdimd32  (decimal32  x, decimal32  y);
  decimal64  fdimd64  (decimal64  x, decimal64  y);
  decimal128 fdimd128 (decimal128 x, decimal128 y);

  decimal32  fmaxd32  (decimal32  x, decimal32  y);
  decimal64  fmaxd64  (decimal64  x, decimal64  y);
  decimal128 fmaxd128 (decimal128 x, decimal128 y);

  decimal32  fmind32  (decimal32  x, decimal32  y);
  decimal64  fmind64  (decimal64  x, decimal64  y);
  decimal128 fmind128 (decimal128 x, decimal128 y);

  // floating multiply-add:
  decimal32  fmad32  (decimal32  x, decimal32  y, decimal32  z);
  decimal64  fmad64  (decimal64  x, decimal64  y, decimal64  z);
  decimal128 fmad128 (decimal128 x, decimal128 y, decimal128  z);

  // 3.6.6.1 abs function overloads
  decimal32  abs(decimal32  d);
  decimal64  abs(decimal64  d);
  decimal128 abs(decimal128 d);
}  }
```

3.6.2 `<cmath>` macros

```
#define HUGE_VAL_D32        implementation-defined
```

Expansion: a positive lvalue of type `decimal32`.

```
#define HUGE_VAL_D64        implementation-defined
```

Expansion: a positive lvalue of type `decimal64`, not necessarily representable as a `decimal32`.

```
#define HUGE_VAL_128        implementation-defined
```

Expansion: a positive lvalue of type `decimal128`, not necessarily representable as a `decimal64`.

```
#define DEC_INFINITY        implementation-defined
```

Expansion: an lvalue of type `decimal32` representing infinity.

```
#define DEC_NAN             implementation-defined
```

Expansion: an lvalue of type `decimal32` representing quiet NaN.

```
#define FP_FAST_FMAD32      implementation-defined
#define FP_FAST_FMAD64      implementation-defined
#define FP_FAST_FMAD128     implementation-defined
```

Effects: these macros are, respectively, `decimal32`, `decimal64`, and `decimal128` analogs of `FP_FAST_FMA` in C99, subclause 7.12.

3.6.3 Evaluation formats

```
typedef decimal-floating-type decimal32_t;
typedef decimal-floating-type decimal64_t;
```

The types `decimal32_t` and `decimal64_t` are decimal floating types at least as wide as `decimal32` and `decimal64`, respectively, and such that `decimal64_t` is at least as wide as `decimal32_t`. If `DEC_EVAL_METHOD` equals 0, `decimal32_t` and `decimal64_t` are `decimal32` and `decimal64`, respectively; if `DEC_EVAL_METHOD` equals 1, they are both `decimal64`; if `DEC_EVAL_METHOD` equals 2, they are both `decimal128`; and for other values of `DEC_EVAL_METHOD`, they are otherwise implementation-defined.

3.6.4 `samequantum` functions

```
bool samequantumd32  (decimal32 x, decimal32 y);
bool samequantumd64  (decimal64 x, decimal64 y);
bool samequantumd128 (decimal128 x, decimal128 y);
```

Effects: determines if the ~~representation~~ quantum exponents of *x* and *y* are the same. If both *x* and *y* are NaN, or infinity, they have the same ~~representation~~ quantum exponents; if exactly one operand is infinity or exactly one operand is NaN, they do not have the same ~~representation~~ quantum exponents. The samequantum functions raise no exception.

Returns: `true` when *x* and *y* have the same representation exponents, `false` otherwise.

```
bool samequantum (decimal32 x, decimal32 y);
```

Returns: `samequantumd32(x, y)`

```
bool samequantum (decimal64 x, decimal64 y);
```

Returns: `samequantumd64(x, y)`

```
bool samequantum (decimal128 x, decimal128 y);
```

Returns: `samequantumd128(x, y)`

3.6.5 `quantexp` functions

```
int quantexpd32  (decimal32 x);
int quantexpd64  (decimal64 x);
int quantexpd128 (decimal128 x);
```

Effects: if *x* is finite, returns its quantum exponent. Otherwise, a domain error occurs and `INT_MIN` is returned.

```
int  quantexp (decimal32 x);
```

Returns: `quantexpd32(x)`

```
int  quantexp (decimal64 x);
```

Returns: `quantexpd64(x)`

```
Int  quantexp (decimal128 x);
```

Returns: `quantexpd128(x)`

3.6.6 `quantized` functions

```
decimal32  quantized32  (decimal32 x, decimal32 y);
decimal64  quantized64  (decimal64 x, decimal64 y);
decimal128 quantized128 (decimal128 x, decimal128 y);
```

Returns: a number that is equal in value (except for any rounding) and sign to *x*, and which has an exponent set to be equal to the exponent of *y*. If the exponent is being increased, the value is correctly rounded according to the current rounding mode; if the result does not have the same value as *x*, the "inexact" floating-point exception is raised. If the exponent is being decreased and the significand of the result has more digits than the type would allow, the "invalid" floating-point exception is raised and the result is NaN. If one or both operands are NaN the result is NaN. Otherwise if only one operand is infinity, the "invalid" floating-point exception is raised and the result is NaN. If both operands are infinity, the result is DEC_INFINITY, with the same sign as *x*, converted to the type of *x*. The quantize functions do not signal underflow.

```
decimal32  quantize (decimal32 x, decimal32 y);
```

Returns: `quantized32(x, y)`

```
decimal64  quantize (decimal64 x, decimal64 y);
```

Returns: `quantized64(x, y)`

```
decimal128 quantize (decimal128 x, decimal128 y);
```

Returns: `quantized128(x, y)`

3.6.6 Elementary functions

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

```
   acos    ceil   floor   log     sin    tanh
   asin    cos    fmod    log10   sinh
   atan    cosh   frexp   modf    sqrt
   atan2   fabs   ldexp   pow     tan
```

```
and for each of the following TR1 elementary functions from <cmath>:
   acosh      expm1    llround     nexttoward
   asinh      fdim     lrint       remainder
   atanh      fma      lround      remquo
   cbrt       fmax     log1p       rint
   copysign   fmin     log2        round
   erf        hypot    logb        scalbn
   erfc       ilogb    nan         scalbln
   exp        lgamma   nearbyint   tgamma
   exp2       llrint   nextafter   trunc
```

- an additional function is introduced to the namespace `std::decimal` with the name *func*`d32`, where *func* is the name of the original function; all parameters of type `double` in the original are replaced with type `decimal32` in the new function; all parameters of type `double *` are replaced with type `decimal32 *`; if the return type of the original function is `double`, the return type of the new function is `decimal32`; the specification of the behavior of the new function is otherwise equivalent to that of the original function

- an additional overload of the original function *func* is introduced to the namespace in which the original is declared; apart from its name and nearest enclosing namespace, this function has the same signature, return type, and behavior as the function *func*`d32`, described above

- an additional function is introduced to the namespace `std::decimal` with the name *func*`d64`, where *func* is the name of the original function; all parameters of type `double` in the original are replaced with type `decimal64` in the new function; all parameters of type `double *` are replaced with type `decimal64 *`; if the return type of the original function is `double`, the return type of the new function is `decimal64`; the specification of the behavior of the new function is otherwise equivalent to that of the original function

- an additional overload of the original function *func* is introduced to the namespace in which the original is declared; apart from its name and nearest enclosing namespace, this function has the same signature, return type, and behavior as the function *func*`d64`, described above

- an additional function is introduced to the namespace `std::decimal` with the name *func*`d128`, where *func* is the name of the original function; all parameters of type `double` in the original are replaced with type `decimal128` in the new function; all parameters of type `double *` are replaced with type `decimal128 *`; if the return type of the original function is `double`, the return type of the new function is `decimal128`; the specification of the behavior of the new function is otherwise equivalent to that of the original function

• an additional overload of the original function *func* is introduced to the namespace in which the original is declared; apart from its name and nearest enclosing namespace, this function has the same signature, return type, and behavior as the function *func*`d128`, described above

Moreover, there shall be additional overloads of the original function *func*, declared in *func*'s namespace, sufficient to ensure:

1. If any argument corresponding to a `decimal64` parameter has type `decimal128`, then all arguments of decimal floating-point type or integer type corresponding to `decimal64` parameters are effectively cast to `decimal128`.

2. Otherwise, if any argument corresponding to a `decimal64` parameter has type `decimal64`, then all other arguments of decimal floating-type or integer-type corresponding to `decimal64` parameters are effectively cast to `decimal64`.

3. Otherwise, if any argument corresponding to a `decimal64` parameter has type `decimal32`, then all other arguments of decimal floating-type or integer-type corresponding to `decimal64` parameters are effectively cast to `decimal32`.

3.6.6.1 `abs` function overloads

```
decimal32  abs(decimal32  d);
decimal64  abs(decimal64  d);
decimal128 abs(decimal128 d);
```

Returns: `fabs(d)`

3.6.7 Changes to `<math.h>`

The header behaves as if it includes the header `<cmath>`, and provides sufficient additional *using* declarations to declare in the global namespace all the additional function and type names introduced by this Technical Report to the header `<cmath>`.

3.6.7.1 Additions to header `<math.h>` synopsis

```
// C-compatibility convenience macros:
#define _Decimal32_t std::decimal::decimal32_t
#define _Decimal64_t std::decimal::decimal64_t
```

3.7 Additions to `<cstdio>` and `<stdio.h>`

This Technical Report introduces the following formatted input/output specifiers for `fprintf`, `fscanf`, and related functions declared in `<cstdio>` and `<stdio.h>`:

```
      H  Specifies that any following e, E, f, F, g, or G conversions
specifier applies to a decimal32 argument.
      D  Specifies that any following e, E, f, F, g, or G conversions
specifier applies to a decimal64 argument.
      DD Specifies that any following e, E, f, F, g, or G conversions
specifier applies to a decimal128 argument.
```

3.8 Additions to `<cstdlib>` and `<stdlib.h>`

3.8.1 Additions to header `<cstdlib>` synopsis

```
namespace std {
namespace decimal {
  // 3.8.2 strtod functions:
  decimal32  strtod32  (const char * nptr, char ** endptr);
  decimal64  strtod64  (const char * nptr, char ** endptr);
  decimal128 strtod128 (const char * nptr, char ** endptr);
}
}
```

3.8.2 `strtod` functions

These functions behave as specified in subclause 9.4 of ISO/IEC TR 24732.

3.8.3 Changes to `<stdlib.h>`

Each name placed into the namespace `decimal` by `<cstdlib>` is placed into both the namespace `decimal` and the global namespace by `<stdlib.h>`.

3.9 Additions to `<cwchar>` and `<wchar.h>`

3.9.1 Additions to `<cwchar>` synopsis

```
namespace std {
namespace decimal {
  // 3.9.2 wcstod functions:
  decimal32  wcstod32  (const wchar_t * nptr, wchar_t ** endptr);
  decimal64  wcstod64  (const wchar_t * nptr, wchar_t ** endptr);
  decimal128 wcstod128 (const wchar_t * nptr, wchar_t ** endptr);
}
}
```

3.9.2 `wcstod` functions

These functions behave as specified in subclause 9.5 of ISO/IEC TR 24732.

3.9.3 Changes to `<wchar.h>`

Each name placed into the namespace `decimal` by `<cwchar>` is placed into both the namespace `decimal` and the global namespace by `<wchar.h>`.

3.10 Facets

This Technical Report introduces the locale facet templates `extended_num_get` and `extended_num_put`. For any locale *`loc`* either constructed, or returned by `locale::classic()`, and any facet *`Facet`* that is one of the required instantiations indicated in Table 3, `std::has_facet<`*`Facet`*`>(`*`loc`*`)` is `true`. Each `std::locale` member function that has a parameter *`cat`* of type `std::locale::category` operates on the these facets when *`cat`* `& std::locale::numeric != 0`.

Table 3 -- Extended Category Facets

Category	Facets

`numeric`	`extended_num_get<char>`, `extended_num_get<wchar_t>`
	`extended_num_put<char>`, `extended_num_put<wchar_t>`

3.10.1 Additions to header `<locale>` synopsis

```
namespace std {
namespace decimal {

  // 3.10.2 extended_num_get facet:
  template <class charT, class InputIterator>
      class extended_num_get;

  // 3.10.3 extended_num_put facet:
  template <class charT, class OutputIterator>
      class extended_num_put;
}
}
```

3.10.2 Class template `extended_num_get`

```
namespace std {
namespace decimal {
  template <class charT,
            class InputIterator = std::istreambuf_iterator<charT,
                std::char_traits<charT> > >
  class extended_num_get : public std::locale::facet {
    public:
      typedef charT char_type;
      typedef InputIterator iter_type;

      explicit extended_num_get(size_t refs = 0);
      extended_num_get(const std::locale & b, size_t refs = 0);

      iter_type get(iter_type in, iter_type end,
                    std::ios_base & str,
                    std::ios_base::iostate & err,
                    decimal32 & val) const;
      iter_type get(iter_type in, iter_type end,
                    std::ios_base & str,
                    std::ios_base::iostate & err,
                    decimal64 & val) const;
      iter_type get(iter_type in, iter_type end,
                    std::ios_base & str,
                    std::ios_base::iostate & err,
                    decimal128 & val) const;

      iter_type get(iter_type in, iter_type end,
                    std::ios_base & str,
                    std::ios_base::iostate & err,
                    bool & val) const;
      iter_type get(iter_type in, iter_type end,
                    std::ios_base & str,
                    std::ios_base::iostate & err,
                    long & val) const;
      iter_type get(iter_type in, iter_type end,
                    std::ios_base & str,
                    std::ios_base::iostate & err,
                    unsigned short & val) const;
      iter_type get(iter_type in, iter_type end,
                    std::ios_base & str,
                    std::ios_base::iostate & err,
                    unsigned int & val) const;
      iter_type get(iter_type in, iter_type end,
                    std::ios_base & str,
                    std::ios_base::iostate & err,
                    unsigned long & val) const;
      iter_type get(iter_type in, iter_type end,
                    std::ios_base & str,
                    std::ios_base::iostate & err,
                    float & val) const;

      iter_type get(iter_type in, iter_type end,
                    std::ios_base & str,
                    std::ios_base::iostate & err,
                    double & val) const;
      iter_type get(iter_type in, iter_type end,
                    std::ios_base & str,
                    std::ios_base::iostate & err,
                    long double & val) const;
      iter_type get(iter_type in, iter_type end,
                    std::ios_base & str,
                    std::ios_base::iostate & err,
                    void * & val) const;

      static std::locale::id id;

    protected:
      ~extended_num_get(); // virtual

      virtual iter_type do_get(iter_type in, iter_type end,
                               std::ios_base & str,
                               std::ios_base::iostate & err,
                               decimal32 & val) const;
      virtual iter_type do_get(iter_type in, iter_type end,
                               std::ios_base & str,
                               std::ios_base::iostate & err,
                               decimal64 & val) const;
      virtual iter_type do_get(iter_type in, iter_type end,
                               std::ios_base & str,
                               std::ios_base::iostate & err,
                               decimal128 & val) const;

      // std::locale baseloc;        exposition only
  };
}
}
```

3.10.2.1 `extended_num_get` members

```
      explicit extended_num_get(size_t refs = 0);
```

Effects: Constructs an `extended_num_get` facet as if by:

```
    typedef std::num_get<charT, InputIterator> base_type;
    explicit extended_num_get(size_t refs = 0)
        : facet(refs), baseloc(std::locale()) { /* ... */ }
```

```
extended_num_get(std::locale & b, size_t refs = 0);
```

Effects: Constructs an `extended_num_get` facet as if by:

```
    extended_num_get(const std::locale & b,                      size_t
refs = 0)
        : facet(refs), baseloc(b) { /* ... */ }
```

```
      iter_type get(iter_type in, iter_type end, std::ios_base & str,
                    std::ios_base::iostate & err,
                    decimal32 & val) const;
      iter_type get(iter_type in, iter_type end, std::ios_base & str,
                    std::ios_base::iostate & err,
                    decimal64 & val) const;
      iter_type get(iter_type in, iter_type end, std::ios_base & str,
                    std::ios_base::iostate & err,
                    decimal128 & val) const;
```

Returns: `do_get(in, end, str, err, val)`.

```
      iter_type get(iter_type in, iter_type end, std::ios_base & str,
                    std::ios_base::iostate & err, bool & val) const;
      iter_type get(iter_type in, iter_type end, std::ios_base & str,
                    std::ios_base::iostate & err, long & val) const;
      iter_type get(iter_type in, iter_type end, std::ios_base & str,
                    std::ios_base::iostate & err,
                    unsigned short & val) const;
      iter_type get(iter_type in, iter_type end, std::ios_base & str,
                    std::ios_base::iostate & err,
                    unsigned int & val) const;
      iter_type get(iter_type in, iter_type end, std::ios_base & str,
                    std::ios_base::iostate & err,
                    unsigned long & val) const;
      iter_type get(iter_type in, iter_type end, std::ios_base & str,
                    std::ios_base::iostate & err, float & val) const;
      iter_type get(iter_type in, iter_type end, std::ios_base & str,
                    std::ios_base::iostate & err, double & val) const;
      iter_type get(iter_type in, iter_type end, std::ios_base & str,
                    std::ios_base::iostate & err,
                    long double & val) const;
      iter_type get(iter_type in, iter_type end, std::ios_base & str,
                    std::ios_base::iostate & err, void * & val) const;
```

Returns: `std::use_facet<std::num_get<charT, InputIterator> >(`*`baseloc`*`).get(`*`in`*`, `*`end`*`, `*`str`*`, `*`err`*`, `*`val`*`)`.

3.10.2.2 `extended_num_get` virtual functions

```
iter_type do_get(iter_type in, iter_type end,
                 std::ios_base & str,
                 std::ios_base::iostate & err,
                 decimal32 & val) const;
iter_type do_get(iter_type in, iter_type end,
                 std::ios_base & str,
                 std::ios_base::iostate & err,
                 decimal64 & val) const;
iter_type do_get(iter_type in, iter_type end,
                 std::ios_base & str,
                 std::ios_base::iostate & err,
                 decimal128 & val) const;
```

Effects: The input characters will be interpreted as described in [lib.facet.num.get.virtuals], and the resulting value will be stored in *val*. For conversions to type decimal32, decimal64, and decimal128, the conversion specifiers are `%gHD`, `%gD`, and `%gLD`, respectively.

Returns: *in*.

3.10.3 Class template `extended_num_put`

```
namespace std {
namespace decimal {
  template <class charT, class OutputIterator =
      std::ostreambuf_iterator<charT, std::char_traits<charT> > >
  class extended_num_put : public std::locale::facet {
    public:
      typedef charT char_type;
      typedef OutputIterator iter_type;

      explicit extended_num_put(size_t refs = 0);
      extended_num_put(const std::locale & b, size_t refs = 0);

      iter_type put(iter_type s, ios_base & f, char_type fill,
                    decimal32 val) const;
      iter_type put(iter_type s, ios_base & f, char_type fill,
                    decimal64 val) const;
      iter_type put(iter_type s, ios_base & f, char_type fill,
                    decimal128 val) const;

      iter_type put(iter_type s, ios_base & f, char_type fill,
                    bool val) const;
      iter_type put(iter_type s, ios_base & f, char_type fill,
                   long val) const;
      iter_type put(iter_type s, ios_base & f, char_type fill,
                    unsigned long val) const;
      iter_type put(iter_type s, ios_base & f, char_type fill,
                    double val) const;
      iter_type put(iter_type s, ios_base & f, char_type fill,
                    long double val) const;
      iter_type put(iter_type s, ios_base & f, char_type fill,
                    const void * val) const;

      static std::locale::id id;

    protected:
      ~extended_num_put();             // virtual

      virtual iter_type do_put(iter_type s, ios_base & f,
                               char_type fill,
                               decimal32 val) const;
      virtual iter_type do_put(iter_type s, ios_base & f,
                               char_type fill,
                               decimal64 val) const;
      virtual iter_type do_put(iter_type s, ios_base & f,
                               char_type fill,
                               decimal128 val) const;

      // std::locale baseloc;        exposition only
  };
}
}
```

3.10.3.1 `extended_num_put` members

```
    explicit extended_num_put(size_t refs = 0);
```

Effects: Constructs an `extended_num_put` facet as if by:

```
explicit extended_num_put(size_t refs = 0)
    : facet(refs), baseloc(std::locale())
    { /* ... */ }

    extended_num_put(std::locale b, size_t refs = 0);
```

Effects: Constructs an `extended_num_put` facet as if by:

```
extended_num_put(const std::locale & b,
                 size_t refs = 0)
: facet(refs), baseloc(b)
{ /* ... */ }

    iter_type put(iter_type s, ios_base & f,
                  char_type fill, decimal32 val) const;
    iter_type put(iter_type s, ios_base & f,
                  char_type fill, decimal64 val) const;
    iter_type put(iter_type s, ios_base & f,
                  char_type fill, decimal128 val) const;
```

Returns: `do_put(s, f, fill, val)`.

```
  iter_type put(iter_type s, ios_base & f,                  char_type
fill, bool val) const;   iter_type put(iter_type s, ios_base & f,
char_type fill, long val) const;   iter_type put(iter_type s, ios_base
& f,                     char_type fill, unsigned long val) const;
iter_type put(iter_type s, ios_base & f,                  char_type
fill, double val) const;   iter_type put(iter_type s, ios_base & f,
char_type fill, long double val) const;   iter_type put(iter_type s,
ios_base & f,                  char_type fill, const void * val) const;
```

Returns: `std::use_facet<std::num_put<chart, OutputIterator> >(baseloc).put(s, f, fill, val)`.

3.10.3.2 `extended_num_put` virtual functions

```
        virtual iter_type do_put(iter_type s, ios_base & f,
                                 char_type fill, decimal32 val) const;
        virtual iter_type do_put(iter_type s, ios_base & f,
                                 char_type fill, decimal64 val) const;
        virtual iter_type do_put(iter_type s, ios_base & f,
                                 char_type fill, decimal128 val) const;
```

Effects: The number represented by *val* will be formatted for output as described in [lib.facet.num.put.virtuals]. A length modifier is added to the conversion specifier as indicated in Table 4.

Table 4 -- Length modifier

type	length modifier
decimal32	HD
decimal64	D
decimal128	LD

Returns: *out*.

3.11 Type traits

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

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

However, the following expression shall yield `true` 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
```

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

The following expression shall yield `true` where *dec* is one of `decimal32`, `decimal64`, or `decimal128`:

```
is_pod<dec>::value
```

3.11.1 Addition to header `<type_traits>` synopsis

```
namespace std {
namespace decimal {
  // 3.11.2 is_decimal_floating_point type_trait:
  template <class T> struct is_decimal_floating_point;
}
}
```

3.11.2 `is_decimal_floating_point` type_trait

`is_decimal_floating_point` is a *UnaryTypeTrait* [tr.meta.rqmts] and satisfies all of the requirements of that category [tr.meta.requirements].

Table 5 -- Type Category Predicates

Template	Condition	Comments
`template <class T> struct is_decimal_floating_point;`	`T` is one of `decimal32`, `decimal64`, or `decimal128`	

3.12 Hash functions

3.12.1 Additions to header `<functional>` synopsis

```
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>;
}
}
```

3.12.2 Hash function specializations

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.

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 incompatibilty inevitable. Differences between the C and C++ decimal types -- and techniques for overcoming them -- are described in this section.

4.1 Use of `<decfloat.h>`

To aid portability to C++, it is recommended that C programmers `#include` the header file `<decfloat.h>` in those translation units that make use of the decimal floating types. This ensures that the equivalent C++ floating-point types will be available, should the program source be ported to C++.

4.2 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.

4.3 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 functions `decimal_to_long_double`, `decimal32_to_long_double`, `decimal64_to_long_double`, and `decimal128_to_long_double` should be used for this purpose. C programmers who wish to maintain portability to C++ should use the `decimal32_to_long_double`, `decimal64_to_long_double`, and `decimal128_to_long_double` forms instead of the cast notation.