Numeric Traits for the Standard Library

Document #: WG21 P0437R1
Date: 2018-11-07
Project: JTC1.22.32 Programming Language C++
Audience: LEWG ⇒ LWG
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

This paper first discusses a vision for a future replacement for the **`<limits>`** header and assorted other related facilities that are today scattered among such other headers as **`<cfloat>`**, **`<climits>`**, and **`<cmath>`**. The paper then proposes a new **`<num_traits>`** header to consolidate and subsume all these facilities.

[Why are numbers beautiful?] It's like asking why is Ludwig van Beethoven's Ninth Symphony beautiful. If you don't see why, someone can't tell you. I know numbers are beautiful. If they aren't beautiful, nothing is.

— PAUL ERDÖS

Two of the most famous Baghdadi scholars, the philosopher Al-Kindi and the mathematician Al-Khawarizmi, were certainly the most influential in transmitting Hindu numerals to the Muslim world. Both wrote books on the subject [that were] translated into Latin and transmitted to the West, thus introducing Europeans to the decimal system, which was known in the Middle Ages only as Arabic numerals. But it would be many centuries before it was widely accepted in Europe. One reason for this was sociological: decimal numbers were considered for a long time as symbols of the evil Muslim foe.

— JAMEEL SADIK "JIM" AL-KHALILI

1 Introduction

The header **`<limits>`** provides **`numeric_limits<>`** and **`enum`**s **`float_round_style`** and **`float_denorm_style`**. The design of this header long predates our modern understanding of traits programming. For example, Meredith and Oram point out in [N2591] that:

> **numeric_limits** is a monolithic class template designed for user-extension through template specialization. Unfortunately this means [that] any attempt to extend the interface by the standard renders all existing user specializations invalid. This problem could have been avoided if each trait in **numeric_limits** had been declared as its own template, such as with the [then-nascent] type traits library.

That paper then proposes to "Break the monolithic **numeric_limits** class template apart into a lot of individual trait templates, so that new traits can be added easily." We certainly agree with this general direction. Moreover, we today have far better tools for accommodating such cleanup than we had at the time (2008). For example, we have:

- **constexpr** and type traits in C++11,
- variable templates and improved type traits in C++14,
- *constexpr if* statements in C++17, and
- *requires-clauses* and *requires-expressions* in the C++20 Working Draft [N4762].

We should also consider breaking changes as we transition to an updated Standard Library. Although our proposal would allow keeping the existing **numeric_limits**, we believe it is a better idea to excise it (and other overlapping features) to avoid maintaining the same information in several forms. Therefore, our proposal will unify and subsume all these and other related facilities.

2 Desiderata

2.1 Short-term

- Create individual numeric traits from the facilities today provided via **<limits>**.
- Collect these new individual traits into a new header, **<num_traits>**, ensuring that each can be user-specialized.

2.2 Long-term

- Identify other numeric library facilities that are traits-like in nature. Provide these facilities (possibily in a different form) in the new **<num_traits>** header, then excise them from their current location.
- Identify and excise other library facilities (e.g., from C headers and their C++ analogs) that duplicate these capabilities.
- In all cases, ensure that no significant capabilities are lost in the process.
- No then-empty header (e.g., **<limits>**) will be retained.

3 Classifying <limits>

Today's **numeric_limits** has 32 members, of several kinds. Some describe characteristics of the given parameter **T**, some provide special values of **T**, etc. All have specified default values, generally **T{}** or **false**. For purposes of exposition, we will classify the members as follows:

1. Applicability:
   - **is_specialized**
2. Distinguished values:
   - **denorm_min()** - **epsilon()** - **infinity()** - **lowest()** - **max()** - **min()**
   - **quiet_NaN()** - **round_error()** - **signaling_NaN()**
3. Value introspection:
   - **has_infinity** - **has_quiet_NaN** - **has_signaling_NaN**

4. Descriptive characteristics:
   • **digits** • **digits10** • **max_digits10** • **max_exponent** • **max_exponent10**
   • **min_exponent** • **min_exponent10** • **radix**
5. Behaviorial characteristics
   • **has_denorm** • **has_denorm_loss** • **is_bounded** • **is_exact** • **is_iec559**
   • **is_integer** • **is_modulo** • **is_signed** • **round_style** • **tinyness_before**
   • **traps**

We will discuss our proposal one classification at a time.

4 Applicability

This classification encompasses but a single member: **is_specialized**. We propose no replacement for this member. To compensate, we will propose, below, that our individual traits be SFINAE-friendly, and will also propose a few general-purpose facilities for detecting values' presence/absence.

5 Distinguished values

This classification encompasses the following **numeric_limits** members: **denorm_min()**, **epsilon()**, **infinity()**, **lowest()**, **max()**, **min()**, **quiet_NaN()**, **round_error()**, and **signaling_NaN()**. With **T** as the template argument, each is specified as a member function returning a value of type **remove_cv_t<T>**.

Given the current state of **constexpr** technology, we believe it is no longer necessary that these be functions. Therefore, we propose that each of these become independent **constexpr** variable templates. We also propose that, as independent traits, these be renamed with a prefix "**num_**":

- **num_denorm_min**
- **num_epsilon**
- **num_infinity**
- **num_lowest**
- **num_max**
- **num_min**
- **num_quiet_NaN**
- **num_round_error**
- **num_signaling_NaN**

We now show a sample implementation for a single numeric trait.[1] We have chosen, somewhat arbitrarily, the **num_infinity** trait (known in **<limits>** as **numeric_limits::infinity()**).

5.1 Exposition-only helpers

We first present a behind-the-scenes implementation of the trait intended for use with only cv-unqualified types. The primary template is defined to provide no result so as to be SFINAE-friendly, while the specializations use GCC-provided intrisics to provide the appropriate results, if any, for floating-point types.

```
1 template< class T > struct __num_infinity_unqual  { };

3 #ifdef __FLT_HAS_INFINITY__
4   template<> struct __num_infinity_unqual<float>
5   { static constexpr float value = __builtin_huge_valf(); };
6 #endif

8 #ifdef __DBL_HAS_INFINITY__
```

[1] Using a recent version of GCC's trunk, we have implemented all the numeric traits in the present proposal. Because all these traits follow a similar coding pattern, we opted to show only one the interest of (at least some) brevity.

```
  template<> struct __num_infinity_unqual<double>
  { static constexpr double value = __builtin_huge_val(); };
#endif

#ifdef __LDBL_HAS_INFINITY__
  template<> struct __num_infinity_unqual<long double>
  { static constexpr long double value = __builtin_huge_vall(); };
#endif
```

5.2 Numeric trait

Here is an implementation for the trait itself. Note that it is crafted to accommodate user-provided specializations, if any, and to employ facility inheritance, otherwise.[2]

```
template< class T >
struct num_infinity : __num_infinity_unqual<T>  { };

template< class T >
  requires is_const_v<T> or is_volatile_v<T>
struct num_infinity<T> : num_infinity< remove_cv_t<T> >  { };
```

5.3 Trait's value

Consistent with C++17's **_v**-suffixed value extraction for type traits and the like, we provide a constrained variable template for the same purpose. If the trait provides no value for a given template argument, the constraint will ensure that the program will be ill-formed. (See §6 for a general means to determine whether such a value exists for a given type.)

```
template< class T >
  requires requires { num_infinity<T>::value; }
constexpr auto num_infinity_v = num_infinity<T>::value;
```

5.4 Backward compatibility

We use a similar technique to provide overloaded function templates to allow value extraction in all cases. The returned value will be the value of the trait, if it exists, or else a user-provided value. (If the user fails to provide such a value, the default value specified for **numeric_limits::infinity()** will be used.)

```
template< class T >
constexpr auto num_infinity_or( T def = T() ) noexcept
{ return def; }

template< class T >
  requires requires { num_infinity<T>::value; }
constexpr auto num_infinity_or( T = T() ) noexcept
{ return num_infinity<T>::value; }
```

We note in passing that the adoption of our paper [P0266R2], combined with C++17's **constexpr if** control structure, has allowed a somewhat more straightforward implementation:

[2] Strictly speaking, no inheritance is needed to implement this specific trait, but this pattern can be used to implement all of the proposed traits. For exposition purposes, we therefore also follow the pattern here.

```
template< class T >
constexpr auto num_infinity_or( T def = T() ) noexcept
{
  if constexpr( requires { num_infinity<T>::value; } )
    return num_infinity<T>::value;
  else
    return def;
}
```

However, we do not propose such a function to accompany each trait. Instead, we propose a single, more general mechanism to guarantee obtaining a value, independent of whether the trait is defined for a given type:

```
template< template< class > class Trait, class T, class R = T >
constexpr R
  value_or( R def = R() ) noexcept
{ return def; }
//
template< template< class > class Trait, class T, class R = T >
  requires requires { Trait<T>::value; }
constexpr R
  value_or( R = R() ) noexcept
{ return Trait<T>::value; }
```

Such a utility permits **numeric_limits**-style backwards compatibility, as follows:

```
constexpr T
  infinity( ) noexcept
{ return value_or<num_infinity, T>(); }
```

6 Value introspection

This classification encompasses the **bool**-valued **numeric_limits** members **has_infinity**, **has_quiet_NaN**, and **has_signaling_NaN**. These are designed to report on the meaningful existence of other members. For example, **numeric_limits<>::has_infinity** is specified as "True if the type has a representation for positive infinity" ([numeric.limits.members]/33).

We believe these traits are neither necessary nor sufficient. Given that our traits are designed to be extensible to program-defined numeric types, is seems useful to have, instead, a more general mechanism to query the existence of any trait for any numeric type. Therefore, we do not propose any one-for-one replacement for the introspective members listed above. Instead, we propose **value_exists**, a single, more general facility, designed to be applicable to all numeric traits and all numeric types.

We see three possible means to provide this **value_exists**.

1. Wait for a proposal from the introspection study group SG7.
2. Rely on the adoption of our paper [P0266R2].
3. Use C++17-style metaprogramming, possibly in combination with C++20 Concepts.

6.1 Implementations

We have implemented and tested **value_exists** as follows:

```
template< template< class > class Trait, class T >
constexpr bool
  value_exists = false;
//
template< template< class > class Trait, class T >
  requires requires { Trait<T>::value; }
constexpr bool
  value_exists<Trait, T> = true;
```

With adoption of our paper [P0266R2], a simpler implementation of **value_exists** became possible:

```
template< template< class > class Trait, class T >
constexpr bool
  value_exists = requires { Trait<T>::value; };
```

6.2 Uses for value_exists

No matter how **value_exists** is provided, here is a typical application, implementing the aforementioned **has_infinity**:

```
template< class T >
constexpr bool
  has_infinity = value_exists<num_infinity, T>;
```

As another use for **value_exists**, here is how it can be applied in C++17 to obtain a more straightforward implementation of **value_or**, described in §5 above:

```
template< template< class > class Trait, class T, class R = T >
constexpr R
  value_or( R def = R() ) noexcept
{
  if constexpr( value_exists<Trait, T> )
    return Trait<T>::value;
  else
    return def;
}
```

Finally, we note that adoption of our paper [P0266R2] can be interpreted as obviating need for this **value_exists** utility. To illustrate this, here is an alternate implementation of our first use case[3]:

```
template< class T >
constexpr bool
  has_infinity = requires { num_infinity<T>::value; };
```

We leave for LEWG the design question of whether we nonetheless want this utility.

7 Descriptive characteristics

This classification encompasses the following members of **numeric_limits**: **digits**, **digits10**, **max_digits10**, **max_exponent**, **max_exponent10**, **min_exponent**, **min_exponent10**, and **radix**. Each of these is specified as **int**-valued. We therefore propose to provide these as independent **constexpr** variable templates. We also propose, as before, that these be renamed with a **num_** prefix:

[3] Of course, the use case itself could be obviated by analogous reasoning.

- `num_digits`
- `num_digits10`
- `num_max_digits10`
- `num_max_exponent`
- `num_max_exponent10`
- `num_min_exponent`
- `num_min_exponent10`
- `num_radix`

As an example, here is an implementation of the proposed `num_max_exponent` trait. Note that it follows the same coding pattern as presented earlier:

```
template< class T > struct max_exponent_unqual  { };
//
template<> struct max_exponent_unqual<float>
{ static constexpr int value = __FLT_MAX_EXP__; };
//
template<> struct max_exponent_unqual<double>
{ static constexpr int value = __DBL_MAX_EXP__; };
//
template<> struct max_exponent_unqual<long double>
{ static constexpr int value = __LDBL_MAX_EXP__; };

template< class T >
struct num_max_exponent : max_exponent_unqual<T>  { };
//
template< class T >
  requires is_const_v<T> or is_volatile_v<T>
struct
  num_max_exponent<T> : num_max_exponent< remove_cv_t<T> >  { };

template< class T >
  requires requires { num_max_exponent<T>::value; }
constexpr auto
  num_max_exponent_v = num_max_exponent<T>::value;
```

8 Behaviorial characteristics

This classification encompasses the following members of `numeric_limits`: `has_denorm`, `has_denorm_loss`, `is_bounded`, `is_exact`, `is_iec559`, `is_integer`, `is_modulo`, `is_signed`, `round_style`, `tinyness_before`, and `traps`. With two exceptions, these members have type `bool`. The exceptions are `has_denorm` (which has type `float_denorm_style`) and `round_style` (whose type is `float_round_style`).

Rather than propose replacements for these characteristics, we recommend that the numerics study group SG6 first analyze the requirements of recent other standards in this area and make recommendations accordingly. We suggest the documents to be examined include IEEE 754-2008 (also known as ISO/IEC/IEEE 60559:2011) on floating-point, the ISO/IEC 10967 series on language-independent computer arithmetic, the relevant C11 annexes, and other relevant recent work by WG14.

SG6 has to date decline to undertake this task, hence our default position is (reluctantly) to recommend traits that retain the flavor of the existing `numeric_limits` members listed above. We will do so in a future paper, assuming favorable disposition of the present proposal.

9 Effects on other headers

The section explores possible ramifications of our proposal with respect to headers inherited from C. Numerous macros are identified as redundant with our proposal. Given the availability of

C++11's **constexpr**, C++14's variable templates, and C++17's **constexpr if**, we believe it is time to reconsider continued support of preprocessor idioms relying on these macros.

9.1 <climits>

This header, inherited from C, consists of macros only. Our proposed traits obviate most of these; the remaining two should go elsewhere. Then support for this header can be deprecated and/or removed outright.

- **CHAR_BIT** — not a numeric trait; move elsewhere as determined by SG6/LEWG
- {**CHAR SCHAR SHRT INT LONG LLONG**}**_MIN** — obviated by **num_min<>**
- **MB_LEN_MAX** — not a numeric trait; move elsewhere as determined by SG6/LEWG
- {**CHAR SCHAR SHRT INT LONG LLONG**}**_MAX** — obviated by **num_max<>**
- {**UCHAR USHRT UINT ULONG ULLONG**}**_MAX** — obviated by **num_max<>**

9.2 <cstdint>

This header, inherited from C, specifies the following macros, among other contents. Each is obviated as shown, and its support should be deprecated and/or removed outright from this header.

- **INT_**[**FAST LEAST**]{**8 16 32 64**}**_MIN** — obviated by **num_min<>**
- [**U**]**INT_**[**FAST LEAST**]{**8 16 32 64**}**_MAX** — obviated by **num_max<>**
- **INT**{**MAX PTR**}**_MIN** — obviated by **num_min<>**
- [**U**]**INT**{**MAX PTR**}**_MAX** — obviated by **num_max<>**
- {**PTRDIFF SIG_ATOMIC WCHAR WINT**}**_MIN** — obviated by **num_min<>**
- {**PTRDIFF SIG_ATOMIC WCHAR WINT**}**_MAX** — obviated by **num_max<>**
- **SIZE_MAX** — obviated by **num_max<>**

9.3 <cfloat>

This header, inherited from C, consists of macros only. Our proposed traits obviate most of them; we would like to do likewise with the rest so that support for this header can be deprecated and/or removed outright.

- **FLT_ROUNDS** — to be determined by SG6/LEWG
- **FLT_EVAL_METHOD** — to be determined by SG6/LEWG
- {**FLT DBL LDBL**}**_HAS_SUBNORM** — to be determined by SG6/LEWG
- **FLT_RADIX** — obviated by **num_radix<>**
- {**FLT DBL LDBL**}**_MANT_DIG** — obviated by **num_digits<>**
- {**FLT DBL LDBL**}**_DECIMAL_DIG** — obviated by **num_digits10<>**
- **DECIMAL_DIG** — obviated by **num_max_digits10<>**
- {**FLT DBL LDBL**}**_DIG** — obviated by **num_digits10<>**
- {**FLT DBL LDBL**}**_MIN_EXP** — obviated by **num_min_exponent<>**
- {**FLT DBL LDBL**}**_MIN_10_EXP** — obviated by **num_min_exponent10<>**
- {**FLT DBL LDBL**}**_MAX_EXP** — obviated by **num_max_exponent<>**
- {**FLT DBL LDBL**}**_MAX_10_EXP** — obviated by **num_max_exponent10<>**
- {**FLT DBL LDBL**}**_MAX** — obviated by **num_max<>**
- {**FLT DBL LDBL**}**_EPSILON** — obviated by **num_epsilon<>**
- {**FLT DBL LDBL**}**_MIN** — obviated by **num_lowest<>**
- {**FLT DBL LDBL**}**_TRUE_MIN** — obviated by **num_min<>**

9.4 `<cmath>`

This header, inherited from C, specifies the following macros, among other contents. Each is obviated as shown, and its support should be deprecated and/or removed outright from this header.

- **`HUGE_VAL[F L]`**— obviated by **`num_infinity<>`**?
- **`INFINITY`** — obviated by **`num_infinity<>`**
- **`NAN`** — obviated by **`num_quiet_NaN<>`**

10 Proposed wording[4]

10.1 For the purposes of SG10 (Feature Testing), no macro seems necessary. As we are proposing an entirely new header, using the standard *has-include-expression* **`__has_include(<num_traits>)`** seems sufficient to detect the availability of this proposal's implementation.

10.2 Edit paragraph 1 of [support.limits.general] as shown.

1 The headers **`<limits>`** (16.3.2), **`<climits>`** (16.3.5), and **`<cfloat>`** (16.3.6) supply characteristics of implementation-dependent arithmetic types (6.7.1). In addition, the header **`<num_traits>`** ([num.traits]) supplies components, collectively known as *numeric traits*, that provide distinguished values and implementation-dependent characteristics of arithmetic types, and that are extensible to provide analogous values and characteristics for program-defined numeric types.

10.3 Add the following new text to clause [support.limits], positioned at the discretion of the Project Editor.

16.3.x Numeric traits [num.traits]

16.3.x.1 General [num.traits.general]

1 Not all traits are applicable to all numeric types, but where a trait is applicable, its definition shall provide a value appropriate to the type's representation. Such values shall be defined so that they are usable as constant expressions. When a trait is instantiated on a cv-qualified type *cv* **`T`**, the trait's value shall be equal to the value, if any, provided by the specialization on the unqualified type **`T`**.

2 Each non-utility trait specified in this subclause is declared as a class template having the following form:

```
template <class T> struct Trait;
```

Each such **`Trait`** shall be specialized, if and as applicable, for each arithmetic type **`T`** ([basic.fundamental]). In addition, a program may specialize each such **`Trait`**, as applicable, for any cv-unqualified, non-reference, program-defined type **`T`**. In all cases, **`is_empty_v<Trait<T>>`** shall be **`true`**.

16.3.x.2 Header `<num_traits>` synopsis [num.traits.syn]

```
namespace std {
  // [num.traits.util], numeric utility traits:
  template <template <class> class Trait, class T>
    constexpr bool value_exists = see below;
```

[4] All proposed additions and ~~deletions~~ are relative to Working Draft [N4762]. Drafting and editorial notes are displayed against a gray background.

```
template <template <class> class Trait, class T, class R = T>
  constexpr R value_or(R def = R()) noexcept;

// [num.traits.val], numeric distinguished value traits:
template <class T> struct num_denorm_min    { see below };
template <class T> struct num_epsilon       { see below };
template <class T> struct num_infinity      { see below };
template <class T> struct num_lowest        { see below };
template <class T> struct num_max           { see below };
template <class T> struct num_min           { see below };
template <class T> struct num_quiet_NaN     { see below };
template <class T> struct num_round_error   { see below };
template <class T> struct num_signaling_NaN { see below };
template <class T>
  constexpr auto num_denorm_min_v = num_denorm_min<T>::value;
template <class T>
  constexpr auto num_epsilon_v = num_epsilon<T>::value;
template <class T>
  constexpr auto num_infinity_v = num_infinity<T>::value;
template <class T>
  constexpr auto num_lowest_v = num_lowest<T>::value;
template <class T>
  constexpr auto num_max_v = num_max<T>::value;
template <class T>
  constexpr auto num_min_v = num_min<T>::value;
template <class T>
  constexpr auto num_quiet_NaN_v = num_quiet_NaN<T>::value;
template <class T>
  constexpr auto num_round_error_v = num_round_error<T>::value;
template <class T>
  constexpr auto num_signaling_NaN_v = num_signaling_NaN<T>::value;

// [num.traits.char], numeric characteristics traits:
template <class T> struct num_digits         { see below };
template <class T> struct num_digits10       { see below };
template <class T> struct num_max_digits10   { see below };
template <class T> struct num_max_exponent   { see below };
template <class T> struct num_max_exponent10 { see below };
template <class T> struct num_min_exponent   { see below };
template <class T> struct num_min_exponent10 { see below };
template <class T> struct num_radix          { see below };
template <class T>
  constexpr auto num_digits_v = num_digits<T>::value;
template <class T>
  constexpr auto num_digits10_v = num_digits10<T>::value;
template <class T>
  constexpr auto num_max_digits10_v = num_max_digits10<T>::value;
template <class T>
  constexpr auto num_max_exponent_v = num_max_exponent<T>::value;
template <class T>
  constexpr auto num_max_exponent10_v = num_max_exponent10<T>::value;
template <class T>
  constexpr auto num_min_exponent_v = num_min_exponent<T>::value;
template <class T>
```

```
    constexpr auto num_min_exponent10_v = num_min_exponent10<T>::value;
  template <class T>
    constexpr auto num_radix_v = num_radix<T>::value;
}
```

16.3.x.3 Numeric utility traits [num.traits.util]

```
template <template <class> class Trait, class T>
  constexpr bool value_exists = see below;
```

1 *Value:* `true` if `Trait<T>` has a member named `value`, and `false` otherwise.

```
template <template <class> class Trait, class T, class R = T>
  constexpr R value_or(R def = R()) noexcept;
```

2 *Returns:* `Trait<T>::value`, if `value_exists<Trait, T>` is `true`; `def`, otherwise.

16.3.x.4 Numeric distinguished value traits [num.traits.val]

1 Each trait in this subclause shall have a `static constexpr T` data member named `value` if and only if the trait is applicable to the type `T`. Such a member, if any, shall be initialized to a value as specified below; otherwise, there shall be no member `value`.

```
template <class T> struct num_denorm_min { see below };
```

2 *Value:* `T`'s minimum positive denormalized value, if any; otherwise, `T`'s minimum positive normalized value, if any.

```
template <class T> struct num_epsilon { see below };
```

3 *Value:* `T(1) - E`, where `E` denotes `T`'s least value greater than 1, if any.

```
template <class T> struct num_infinity { see below };
```

4 *Value:* `T`'s positive infinity, if any.

```
template <class T> struct num_lowest { see below };
```

5 *Value:* A finite value x, if any, such that `T` has no other finite value y such that $y < x$.

```
template <class T> struct num_max { see below };
```

6 *Value:* A finite value x, if any, such that `T` has no other finite value y such that $y > x$.

```
template <class T> struct num_min { see below };
```

7 *Value:* `T`'s minimum positive normalized value, if `T` supports denormalization; otherwise, the same value as `num_lowest`, if any.

```
template <class T> struct num_quiet_NaN { see below };
```

8 *Value:* `T`'s quiet "Not a Number" value, if any.

```
template <class T> struct num_round_error { see below };
```

9 *Value:* `T`'s maximum rounding error, if any.

```
template <class T> struct num_signaling_NaN { see below };
```

10 *Value:* `T`'s signaling "Not a Number" value, if any.

16.3.x.5 Numeric characteristics traits [num.traits.char]

1 Each trait in this subclause shall have a **static constexpr int** data member named **value** if and only if the trait is applicable to the type **T**. Such a member, if any, shall be initialized to a value as specified below; otherwise, there shall be no member **value**.

```
template <class T> struct num_digits { see below };
```

2 *Value:* The number of **num_radix_v<T>** digits that can be represented without change. If **is_integer_v<T>** is **true**, this is the number of non-sign bits in the representation; If **is_floating_point_v<T>** is **true**, this is the number of **num_radix_v<T>** digits in the mantissa.

```
template <class T> struct num_digits10 { see below };
```

3 *Value:* The number of base 10 digits that can be represented without change.

```
template <class T> struct num_max_digits10 { see below };
```

4 *Value:* The number of base 10 digits required to ensure that type **T** values which differ are always differentiated.

```
template <class T> struct num_max_exponent { see below };
```

5 *Value:* With $r =$ **num_radix_v<T>**, the largest positive integer i such that r^{i-1} is a representable finite value of type **T**.

```
template <class T> struct num_max_exponent10 { see below };
```

6 *Value:* The largest positive integer i such that 10^i is in the range of representable finite type **T** values.

```
template <class T> struct num_min_exponent { see below };
```

7 *Value:* With $r =$ **num_radix_v<T>**, the minimum negative integer i such that r^{i-1} is a representable, normalized (if applicable), finite value of type **T**.

```
template <class T> struct num_min_exponent10 { see below };
```

8 *Value:* The minimum negative integer i such that 10^i is in the range of representable, normalized (if applicable), finite type **T** values.

```
template <class T> struct num_radix { see below };
```

9 *Value:* The base of the representation. If **is_floating_point_v<T>** is **true**, this shall refer to the base or radix (often 2) of the exponent representation.

11 Acknowledgments

Many thanks to the readers of early drafts of this paper for their thoughtful comments.

12 Bibliography

[N2591] Meredith, Alisdair and Fabien Oram: "Refactoring **numeric_limits**." ISO/IEC JTC1/SC22/WG21 document N2591 (post-Bellevue mailing), 2008-03-17. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2591.html.

[N4762] Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4762 (corrected post-Rappersville mailing), 2018-07-07. http://wg21.link/n4762.

[P0266R2] Walter E. Brown: "Lifting Restrictions on requires-Expressions." ISO/IEC JTC1/SC22/WG21 document P0266R2 (post-Issaquah mailing), 2016-11-10. Online: http://wg21.link/p0266r2.

13 Document history

Version	Date	Changes
0	2016-10-14	• Published as P0437R0 (pre-Issaquah).
0	2018-11-07	• Rebased on [N4762]. • Excised wording to remove headers `<limits>`, `<cfloat>`, and `<climits>`. • Published as P0437R1 (pre-San Diego).