Wording for Individually Specializable Numeric Traits

Document #: WG21 P1841R1
Date: 2020–05–03
Audience: LWG
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

This paper combines/integrates/consolidates the wording from its predecessor papers [P0437R1] and [P1370R1].

[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 ERDŐS

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 during al-Ma'mun's reign, and it was their work that was 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

I'm writing a book. I've got the page numbers done.

— STEVEN WRIGHT

1 Introduction

This paper combines/integrates/consolidates the wording from its predecessor papers [P0437R1] and [P1370R1], each of which papers have been favorably reviewed by SG6 and LEWG. While those papers were originally approved for C++20, we opted to retard their progress due to LWG's huge backlog at the time. We now submit their combined proposed wording, adjusted per LEWG's review comments, for LWG review early in the C++23 cycle. Please see those earlier papers[1] for motivation, background information, examples, and other discussion of the present proposal.

[1] There are no plans to revise either of those predecessor papers.

2 Proposed wording[2,3]

2.1 After adjusting ***yyyymm*** (below) so as to denote this proposal's month of adoption, insert the following line among the similar directives following [version.syn]/2:

```
#define __cpp_lib_numeric_traits yyyymmL // also in <numbers>
```

2.2 Edit [numerics.general] as shown.

2 The following subclauses describe components for complex number types, random number generation, numeric (*n*-at-a-time) arrays, generalized numeric algorithms, traits for numeric types, and mathematical constants and functions for floating-point types, as summarized in Table [tab:numerics.summary].

2.3 Edit [support.limits.general] as shown.

1 The headers **`<limits>`** (17.3.3), **`<climits>`** (17.3.6), and **`<cfloat>`** (17.3.7) supply implementation-dependent characteristics of ~~implementation-dependent~~ arithmetic types (6.8.1). In addition, the header **`<numbers>`** ([num.traits]) supplies components, collectively known as *numeric traits*, that not only provide implementation-dependent distinguished values and characteristics for arithmetic types, but that are (via specialization) individually extensible to provide analogous values and characteristics for program-defined numeric types.

2.4 Edit [basic.fundamental], splitting the existing paragraph 12 where shown, updating the new paragraph 13's text as shown, and then incrementing the subsequent paragraph numbers in the subclause. [*Note*: The intent of these changes has been previously discussed with the Project Editor, who approved and contributed much of the following new wording. —*end note*]

12 There are three *floating-point types*: **`float`**, **`double`**, and **`long double`**.

⋮

[*Note*: This document imposes no requirements on the accuracy of floating-point operations; see also 17.3. —*end note*]

13 Integral and floating-point types are collectively called *arithmetic* types. ~~Specializations of the standard library template **`std::numeric_limits`** (17.3) shall specify the maximum and minimum values of each arithmetic type for an implementation.~~ [*Note*: Properties of the arithmetic types, such as their minimum and maximum representable value, can be queried using the facilities in the standard library headers **`<limits>`** ([limits.syn]), **`<climits>`** ([climits.syn]), **`<cfloat>`** ([cfloat.syn]), and **`<numbers>`** ([num.traits.syn]). —*end note*]

~~13~~14 A type *cv* **`void`** is . . . .

~~14~~15 A value of type **`std::nullptr_t`** is . . . .

~~15~~16 The types described in this subclause are . . . .

[2] Proposed additions (entirely new subclauses are colored but not underlined) and ~~deletions~~ are based on [N4861]. Editorial instructions and drafting notes look like this.

[3] The LaTeX source of this wording is available to the Project Editor upon request.

2.5 Add the following new text to clause [support.limits], positioned at the discretion of the Project Editor. [*Note*: Much of the wording specifying the individual traits is taken or adapted from the corresponding **`numeric_limits`** wording. —*end note*]

17.3.x Numeric traits [num.traits]

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

1 Not all numeric traits are applicable to all numeric types, but where a numeric trait is applicable, its definition provides a value appropriate to the type's representation. Such values are defined so that they are usable as constant expressions. When a numeric 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 Except for those in [num.traits.util], each numeric trait specified in this subclause [num.traits] is declared as a class template of the following form:

```
template <class T> struct TRAIT { };
```

Each such primary template is explicitly specialized, if and as applicable, for each arithmetic type ([basic.fundamental]). In addition, a program may explicitly specialize each such primary template for any program-defined numeric type ([numeric.requirements]) to which the numeric trait is applicable.

3 Each specialization ***`TRAIT`*`<T>`** has no members other than a **`static inline constexpr`** data member named **`value`** that is initialized to a value consistent with ***`TRAIT`*`<T>`**'s specification. [*Note*: If there is no explicit specialization ***`TRAIT`*`<T>`** for a given **`T`**, there is no member **`value`**. —*end note*]

17.3.x.2 Header `<numbers>` synopsis [num.traits.syn]

```
namespace std::numbers {

  // [num.traits.util], numeric utility traits
  template <template <class> class Trait, class T>
    inline constexpr bool value_exists = see below;
  template <template <class> class Trait, class T, class R = T>
    inline constexpr R value_or(R def = R()) noexcept;

  // [num.traits.val], numeric distinguished value traits
  template <class T> struct denorm_min                   { see below };
  template <class T> struct epsilon                      { see below };
  template <class T> struct finite_max                   { see below };
  template <class T> struct finite_min                   { see below };
  template <class T> struct infinity                     { see below };
  template <class T> struct norm_min                     { see below };
  template <class T> struct quiet_NaN                    { see below };
  template <class T> struct reciprocal_overflow_threshold { see below };
  template <class T> struct round_error                  { see below };
  template <class T> struct signaling_NaN                { see below };

  template <class T>
    inline constexpr auto denorm_min_v = denorm_min<T>::value;
  template <class T>
    inline constexpr auto epsilon_v = epsilon<T>::value;
  template <class T>
    inline constexpr auto finite_max_v = finite_max<T>::value;
```

```
  template <class T>
    inline constexpr auto finite_min_v = finite_min<T>::value;
  template <class T>
    inline constexpr auto infinity_v = infinity<T>::value;
  template <class T>
    inline constexpr auto norm_min_v = norm_min<T>::value;
  template <class T>
    inline constexpr auto quiet_NaN_v = quiet_NaN<T>::value;
  template <class T>
    inline constexpr auto reciprocal_overflow_threshold_v
      = reciprocal_overflow_threshold<T>::value;
  template <class T>
    inline constexpr auto round_error_v = round_error<T>::value;
  template <class T>
    inline constexpr auto signaling_NaN_v = signaling_NaN<T>::value;

  // [num.traits.char], numeric characteristics traits
  template <class T> struct digits         { see below };
  template <class T> struct digits10       { see below };
  template <class T> struct max_digits10   { see below };
  template <class T> struct max_exponent   { see below };
  template <class T> struct max_exponent10 { see below };
  template <class T> struct min_exponent   { see below };
  template <class T> struct min_exponent10 { see below };
  template <class T> struct radix          { see below };

  template <class T>
    inline constexpr auto digits_v = digits<T>::value;
  template <class T>
    inline constexpr auto digits10_v = digits10<T>::value;
  template <class T>
    inline constexpr auto max_digits10_v = max_digits10<T>::value;
  template <class T>
    inline constexpr auto max_exponent_v = max_exponent<T>::value;
  template <class T>
    inline constexpr auto max_exponent10_v = max_exponent10<T>::value;
  template <class T>
    inline constexpr auto min_exponent_v = min_exponent<T>::value;
  template <class T>
    inline constexpr auto min_exponent10_v = min_exponent10<T>::value;
  template <class T>
    inline constexpr auto radix_v = radix<T>::value;
}
```

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

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

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

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

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

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

1 The `value` member of the traits specified in this subclause has type `T`.

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

2 *Value:* `T`'s minimum positive denormalized value, if any; otherwise, the same value as `min_normal_v<T>`, if any.

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

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

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

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

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

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

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

7 *Value:* `T`'s minimum positive normalized value, if `T` supports subnormal numbers; otherwise, `T`'s minimum positive value.

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

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

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

9 *Value:* The smallest positive value x of type `T` such that `T(1)`$/x$ does not overflow.

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

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

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

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

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

1 The `value` member of the traits specified in this subclause has type `int`.

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

2 *Value:* The number of `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 `radix_v<T>` digits in the mantissa.

```
template <class T> struct digits10 { see below };
```

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

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

5 *Value:* With $r =$ `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 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 min_exponent { see below };
```

7 *Value:* With $r =$ `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 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 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.

3 Acknowledgments

Many thanks to the readers of early drafts of this paper for their careful proofreading. Special thanks to Mark Hoemmen and Damien Lebrun-Grandie, authors of [P1370R1], for their support of this proposal and their contributions to it.

4 Bibliography

[N4820] Richard Smith: “Working Draft, Standard for Programming Language C++.” ISO/IEC JTC1/SC22/WG21 document N4820 (pre-Cologne mailing), 2019–06–17. https://wg21.link/n4820.

[N4861] Richard Smith: “Working Draft, Standard for Programming Language C++.” ISO/IEC JTC1/SC22/WG21 document N4861 (2020-04 mailing), 2020–04–01. https://wg21.link/n4861.

[P0437R1] Walter E. Brown: “Numeric Traits for the Standard Library.” ISO/IEC JTC1/SC22/WG21 document P0437R1 (pre-San Diego mailing), 2018–11–07. https://wg21.link/p0437r1.

[P1370R1] Mark Hoemmen and Damien Lebrun-Grandie: “Generic numerical algorithm development with(out) `numeric_limits`.” ISO/IEC JTC1/SC22/WG21 document P1370R1 (post-San Diego mailing), 2019–03–10. https://wg21.link/p1370r1.

5 Index of library names

6 Document history

Rev.	Date	Changes
0	2019–08–02	• Published as P1841R0, post-Cologne mailing.
1	2020–05–03	• Rebased on [N4861]. • Applied several wording improvements. • Published as P1841R1, 2020-05 mailing.