Document Number:	P2551R2
Date:	2022-06-22
Reply-to:	Matthias Kretz <m.kretz@gsi.de> Jonathan Wakely <cxx@kayari.org>
Audience:	LEWG
Target:	C++23

Clarify intent of P1841 numeric traits

ABSTRACT

A list of design-related questions after implementation of [P1841R2] “Wording for Individually Specializable Numeric Traits”.

CONTENTS

0 Changelog

0.1 changes from revision 0

Previous revision: P2551R0

- Removed questions that were answered in the last telecon.
- Present options for `reciprocal_overflow_threshold`.

0.2 changes from revision 1

Previous revision: P2551R1

- Back to presenting all questions
- Added section “Proposed Changes”

1 Introduction

[P1841R2] provides wording for numeric traits. The last design paper was [P0437R1] with additions from [P1370R1].

2 Design Questions

1. When exactly is a trait disabled for a given numeric type? It seems the intent was for the `value` member to be defined whenever a representation for the desired constant exists. The wording needs to clarify whether any behavioral aspects play a role. For example, a `denorm_min` may be enabled independent of whether the execution environment flushes denormals to zero / treats denormals as zero. Even in the case of a processor that unconditionally zeros denormals; as long as a representation exists, is the trait enabled? Conversely, if a representation does not exist, is the trait disabled? Specifically, `denorm_min` should never have the value of `norm_min`?
2. Please clarify whether we want to treat `bool` as a numeric type and enable the traits accordingly. The current wording in [P1841R2] enables the traits for `bool`, which is consistent with `std::numeric_limits`. `std::numeric_limits<bool>` will still exist if needed. Numeric code does not use `bool` as a numeric type, despite it being technically an “arithmetic type” in the core language.

3. Many of the numeric traits are motivated by floating-point and make little sense for integral types. Is it intended that all of the following numeric traits are enabled also for integral types?
   - `denorm_min`
   - `epsilon`
   - `max_exponent`
   - `max_exponent10`
   - `min_exponent`
   - `min_exponent10`
   - `infinity`
   - `quiet_NaN`
   - `signaling_NaN`
4. `reciprocal_overflow_threshold` is currently defined as:

P1841R2 [num.traits.val]

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

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

This yields a subnormal number for IEC559 types. How should this value change wrt. treat-denormals-as-zero? I.e. in a situation where the hardware treats subnormal operands as zero you get 1/0 -> inf, which does overflow. In which case it doesn't match the specification anymore. This trait is specified by a behavior and as such may depend on processor state. As a compile-time constant this value must be independent from runtime behavior. But what is the correct value? See `https://godbolt.org/z/eWxdnTYf8` for a demonstration of the problem.

Update after consultation with Mark and Damien (the P1370R1 authors):

- It would be possible to decouple the specification from runtime behavior by specifying behavior of constant expressions only; i.e. that `T(1)`$/x$ does not overflow *in a constant expression*.
- P1370R1 presented an algorithm to determine the value and it does not yield the "*smallest* positive value x of type `T` such that `T(1)`$/x$ does not overflow".

- The P1370R1 algorithm seems to ensure that the value is never subnormal. Thus, the specification should have been "The smallest positive *normal* value x of type `T` such that `T(1)`$/x$ does not overflow"
- Since the actual reciprocal overflow threshold depends on runtime state, we're not sure who would/should use a compile-time constant. It seems simpler and safer to remove `reciprocal_overflow_threshold` from P1841. Mark wrote:

  > I would prefer to remove `reciprocal_overflow_threshold` entirely. The intent of the feature was to describe actual computer behavior at run time, so that library authors could write generic code. However, we can't do that with traits. For example, traits can't change value based on compiler flags. I wish I had realized that better when proposing the feature.

5. `numeric_limits::max_digits10` is 0 for integral types. Is `max_digits10_v<int>` supposed to yield `digits10_v<int> + 1`? Or should it only be specialized for floating-point?

3 PROPOSED CHANGES

After reviewing P2551, Library Evolution wanted to make the following changes:

1. Allow deviation of new individually specializable numeric traits from `numeric_limits`.
2. Base new individually specializable numeric traits on representation rather than behavior.
3. Disable `bool` for new individually specializable numeric traits.
4. Disable the new individually specializable numeric traits for integral types when they are not meaningful for `numeric_limits`.
5. `max_digits10` should be enabled for integral types (yielding `digits_10_v + 1`).
6. Remove `reciprocal_overflow_threshold`.

4 STRAW POLLS

4.1 LEWG telecon 2022-03-29

Poll: Numeric traits can deviate from `numeric_limits`.

SF	F	N	A	SA
13	8	0	0	0

Poll: Numeric traits should be based on representation rather than behavior (ignoring `reciprocal_overflow_threshold`).

SF	F	N	A	SA
7	5	2	0	0

Poll: All numeric traits for bool should be disabled.

SF	F	N	A	SA
12	6	1	0	0

Poll: The numeric traits that are not meaningful for `numeric_limits` (`denorm_min`, `epsilon`, etc) should be disabled for integral types.

SF	F	N	A	SA
14	3	0	0	0

Poll: `max_digits10` should deviate from `numeric_limits` and yields `digits10_v + 1`.

SF	F	N	A	SA
6	5	2	0	0

4.2 LEWG telecon 2022-06-07

Poll: Remove `reciprocal_overflow_threshold` from P1841.

SF	F	N	A	SA
6	4	1	0	0

A BIBLIOGRAPHY

[P0437R1] Walter E. Brown. *P0437R1: Numeric Traits for the Standard Library*. ISO/IEC C++ Standards Committee Paper. 2018. URL: `https://wg21.link/p0437r1`.

[P1841R2] Walter E. Brown. *P1841R2: Wording for Individually Specializable Numeric Traits*. ISO/IEC C++ Standards Committee Paper. 2021. URL: https://wg21.link/p1841r2.

[P1370R1] Mark Hoemmen and Damien Lebrun-Grandie. *P1370R1: Generic numerical algorithm development with(out) numeric_limits*. ISO/IEC C++ Standards Committee Paper. 2019. URL: https://wg21.link/p1370r1.