p0647r1 - Floating point value access for std::ratio

Peter Sommerlad

2017-07-21

Document Number:	p0647r1
Date:	2017-07-21
Project:	Programming Language C++
Audience:	LEWG/LWG/SG6(numerics)
Target:	C++20

1 Motivation

Preparing for standardizing units and using `std::ratio` for keeping track of fractions often one needs to get the quotient as a floating point number or as a number of a type underlying a quantity, e.g., a fixpoint type. Doing that manually means adding a cast before doing the division. This is tedious and it would be nice to just access the value, as one can do with std::integral_constant. I believe that omission is just a historical accident, because it was not possible to do compile-time computation with floating point values when ratio was invented. There are some options on how to access the fraction as a compile-time entity. In the first revision I chose to make the value member a long double and provide a templatized explicit conversion operator for accessing the fraction.

However, SG6 was not completely happy and suggested a variable template instead, which would require to use `template` in front of the variable in deduced contexts, that others didn't like. Therefore, I decided to think a bit more about it and provide some more alternatives with their individual advantages and disadvantages and hope LEWG is able to judge and guide further direction. The ultimate connection between compile-time/run-time-like values would be to go into the direction of boost::hana, by providing operator overloads to allow computation with ratio values (at compile time) to determine the ratio template instance in the end. One can even provide a constructor taking two `integral_constant`s with a deduction guide to guarantee that only simplified ratios would be created.

A further observation by SG6 was, that most of the time you need the quotient of a ratio for further computation, you want to scale a factor. At least that is the case in the units library that is my use case. Therefore, I also provide the alternative for a scale member function that would allow slightly more precise computation, by first multiplying the numerator with the factor and then dividing by the denominator. A problem might be integral factors that can lead to integral overflow raising undefined behavior, but I consider that use case rare.

1.1 Questions to LEWG

- Is the `scale()` static member function template approach OK? Is the name appropriate?
- Are any other options worth considering, esp. moving forward towards value-based computation?

2 Design Options

This section lists several of the non-exclusive options on implementation. LEWG can easily pick-and chose from them, where I consider the `scale(factor)` function essential, all other things are optional.

2.1 Scaling a factor

Add the following static constexpr member function to the `std::ratio` class template:

```
template<typename NUMERIC> //requires multiplication, division, and conversion from intmax_t
static constexpr
    auto scale(NUMERIC factor) {
    return (factor*static_cast<NUMERIC>(num))/static_cast<NUMERIC>(den);
}
```

Consequences:

- This is the most common use case.
- Can increase precision by first multiplying, risking integral overflow.
- Can lead to integer division surprises if `factor` is an integral value.
- Division by zero is a non-issue, there can not be such a `ratio` type.

2.2 Access to quotient through a variable template

This was suggested by SG6. Add the following member variable template to the `std::ratio` class template:

```
template<typename NUMERIC>
static inline constexpr NUMERIC quotient{
    static_cast<NUMERIC>(num)/static_cast<NUMERIC>(den)
};
```

Consequences:

- Allows quotient access in any type of user's choice that allows division
- Can lead to integer division surprises if `NUMERIC` is an integral type.
- A big potential disadvantage is the need to use template keyword in a deduced context: `std::ratio<N,D>::template quotient<double>` to access the quotient. This could be very common in a units library, that makes intensive use of templates.
- Should not be the only option to access the quotient.

2.3 Access to quotient as a long double

This was rejected by SG6. Add the following static constexpr inline member variable to the `std::ratio` class template:

```
static inline constexpr
long double value { static_cast<long double>(num)/den };
```

Consequences:

- uses the type implied for compile-time floating point literals giving the most precision
- SG6 noted that the name value should be reserved for a future, where we have an exact means to represent arbitrary fractions at run-time (beyond an `intmax_t` pair).
- if this is the only means of value access would prohibit using fixed point types for fractions easily (they would do what everyone needs to do with `ratio` today).

2.4 A generic explicit type conversion operator

This was suggested in R0 and is repeated here, because it will make the use of `std::ratio` value objects (that are empty), like boost::hana would do very rational. Whenever you need the quotient, just `static_cast` the ratio value to the type you need the quotient in. However, to make that use more effective, would require to provide a full set of operators working on ratio values (and best also `integral_constant`), leading close to what boost::hana is doing in that area.

```
template<typename NUMERIC>
explicit constexpr operator NUMERIC() const{
     return static_cast<NUMERIC>(num)/static_cast<NUMERIC>(den);
}
```

Consequences:

- Usage requires object instances of `std::ratio` types and static_cast, i.e. `static_cast<double>(std::ratio<42,5>{})`
- Allows quotient access in any type of user's choice that allows division
- Can lead to integer division surprises if `NUMERIC` is an integral type.
- Should not be the only option to access the quotient.
- Only makes real sense, when creation and computation of `std::ratio` values is syntactically pleasing.

2.5 Moving `ratio` towards value-based computation

For the sake of completeness one might consider adding the following operators to the `<ratio>` header.

```
template<intmax_t N1,intmax_t D1,intmax_t N2,intmax_t D2>
constexpr auto operator+(std::ratio<N1,D1>,std::ratio<N2,D2>){
        return std::ratio_add<std::ratio<N1,D1>,std::ratio<N2,D2>>{};
}
template<intmax_t N1,intmax_t D1,intmax_t N2,intmax_t D2>
constexpr auto operator-(std::ratio<N1,D1>,std::ratio<N2,D2>){
        return std::ratio_subtract<std::ratio<N1,D1>,std::ratio<N2,D2>>{};
```

```
}
template<intmax_t N1,intmax_t D1,intmax_t N2,intmax_t D2>
constexpr auto operator*(std::ratio<N1,D1>,std::ratio<N2,D2>){
        return std::ratio_multiply<std::ratio<N1,D1>,std::ratio<N2,D2>>{};
}
template<intmax_t N1,intmax_t D1,intmax_t N2,intmax_t D2>
constexpr auto operator/(std::ratio<N1,D1>,std::ratio<N2,D2>){
        return std::ratio_divide<std::ratio<N1,D1>,std::ratio<N2,D2>>{};
}
template<intmax_t N1,intmax_t D1>
constexpr auto operator-(std::ratio<N1,D1>){
        return std::ratio<-N1,D1>{};
}
template<intmax_t N1,intmax_t D1,intmax_t N2,intmax_t D2>
constexpr auto operator==(std::ratio<N1,D1>,std::ratio<N2,D2>){
        return std::ratio_equal<std::ratio<N1,D1>,std::ratio<N2,D2>>{};
}
template<intmax_t N1,intmax_t D1,intmax_t N2,intmax_t D2>
constexpr auto operator!=(std::ratio<N1,D1>,std::ratio<N2,D2>){
        return std::ratio_not_equal<std::ratio<N1,D1>,std::ratio<N2,D2>>{};
}
template<intmax_t N1,intmax_t D1,intmax_t N2,intmax_t D2>
constexpr auto operator<(std::ratio<N1,D1>,std::ratio<N2,D2>){
        return std::ratio_less<std::ratio<N1,D1>,std::ratio<N2,D2>>{};
}
template<intmax_t N1,intmax_t D1,intmax_t N2,intmax_t D2>
constexpr auto operator<=(std::ratio<N1,D1>,std::ratio<N2,D2>){
        return std::ratio_less_equal<std::ratio<N1,D1>,std::ratio<N2,D2>>{};
}
template<intmax_t N1,intmax_t D1,intmax_t N2,intmax_t D2>
constexpr auto operator>(std::ratio<N1,D1>,std::ratio<N2,D2>){
        return std::ratio_greater<std::ratio<N1,D1>,std::ratio<N2,D2>>{};
}
template<intmax_t N1,intmax_t D1,intmax_t N2,intmax_t D2>
constexpr auto operator>=(std::ratio<N1,D1>,std::ratio<N2,D2>){
        return std::ratio_greater_equal<std::ratio<N1,D1>,std::ratio<N2,D2>>{};
}
```

2.5.1 ratio values from integral_constant

For completeness of value-based computation one can add the following constructors to std::ratio together with a deduction guide guaranteeing only simplified instances.

```
        constexpr ratio()noexcept=default;
        template<intmax_t _Num_,intmax_t _Den_>
        constexpr ratio(std::integral_constant<intmax_t,_Num_>,std::integral_constant<intmax_t,_Den_>)n
                    static_assert(_Num==this->num,"should always be simplified");
                    static_assert(_Den==this->den,"should always be simplified");
        }
  //... and outside the deduction guide: (using libstdc++'s internals:)
  template <intmax_t _Num_, intmax_t _Den_>
```

```
ratio(std::integral_constant<intmax_t,_Num_>,std::integral_constant<intmax_t,_Den_>)
  ->ratio<_Num_ * __static_sign<_Den_>::value / __static_gcd<_Num_, _Den_>::value,
          __static_abs<_Den_>::value / __static_gcd<_Num_, _Den_>::value>;
```

To make this useful a UDL suffix operator converting integral literals to `std::integral_constant<intmax_t,N>` is useful, like boost::hana's `operator"" _c()` leading to the ability to write code like: [*Example:*

```
using namespace std::literals;
using std::ratio;
constexpr ratio r{2_to_c,4_to_c};
constexpr auto fourth=r*r;
static_assert(std::is_same_v<ratio<1,2>,decltype(r)>,"argument deduction wrong");
static_assert(ratio<1>{}==ratio<1,2>{}+r,"add failed");
static_assert(ratio<1,4>{}==fourth,"mul failed");
ASSERT_EQUAL((ratio<1,4>{}),fourth);
```

— *end example*]

3 Changes from R0

SG6 in Toronto suggested to drop the template explicit conversion operator and make the accessor a variable template instead. Also the name `value` should be reserved for future arbitrary precision rational number type in the std to keep the quotient.

It was noted that variable template will require `template` in front of `ratio<a,b>::template quotient<>`.

Also someone suggested to provide a scale static member function template, because that is the most common use case and allow to deduce the floating point type to be used. I have chosen to suggest this variant, but also prepared other options for LEWG to consider for further guidance.

4 Acknowledgements

— Authors of N2661: Howard Hinnant, Walter Brown, Jeff Garland, Marc Paterno.

— Members of the LiaW workshop "Towards Units" at C++Now 2017: Billy Baker, Charles Wilson, Daniel Pfeifer, Dave Jenkins, Manuel Bergler, Morris Hafner, Nicolas Holthaus, Peter Bindels, Steven Watanabe, Tuan Tran.

— Participants of SG6 in Toronto, mainly Walter Brown for leading that group's discussion.

5 Changes Proposed

Modify section 23.26.3 by inserting access to the fractional value represented through a `scale` function.

5.0.1 Class template `ratio` [ratio.ratio]

```
namespace std {
  template <intmax_t N, intmax_t D = 1>
  class ratio {
  public:
    static constexpr intmax_t num;
    static constexpr intmax_t den;

    using type = ratio<num, den>;

    template<typename R>
    static constexpr auto
    scale(R factor) { return (factor * static_cast<R>(num)) / static_cast<R>(den) ; }

  };
}
```

2 Add a new paragraph 3 to the section with the following example:

3 [*Example:* The `scale` static member function template can be used to scale a factor by the quotient represented by a `ratio`:

```
assert(3e12 == std::tera::scale(3.));
assert(10 == std::deci::scale(100));
```

— *end example*]