Reducing `<ratio>`

Document #: WG21 P0656R0
Date: 2017-06-11
Project: JTC1.22.32 Programming Language C++
Audience: SG6 ⇒ LEWG ⇒ LWG
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

Using header `<ratio>` today, the ratio 1 : 2 could be represented by any of a potentially large number of distinct types: `ratio<1,2>`, `ratio<2,4>`, `ratio<5,10>`, etc. Algorithms must therefore be constantly aware that any particular `ratio` type may be in *unreduced* form.

As promised in [P0548R0], this paper proposes to adjust the specification of those `<ratio>` types corresponding to unreduced ratios so that they become aliases for the unique type corresponding to that ratio in *reduced* form. (E.g., `ratio<2,4>` and `ratio<5,10>` would each alias `ratio<1,2>`.) This will reduce cognitive burden on programmers and can also lead to somewhat simpler code.

The ratio of something to nothing is infinite. So just do something.

— PETER DIAMANDIS

The only true measure of success is the ratio between what we might have done and what we might have been on the one hand, and the thing we have made and the things we have made of ourselves on the other.

— HERBERT GEORGE "H.G." WELLS

There's a fine line between a numerator and a denominator. (Only a fraction of people will find this funny.)

— UNKNOWN

1 Background and proposal

Elementary school children are routinely taught how to *reduce* a fraction to its lowest terms: divide both its numerator and its denominator by their greatest common divisor. Children are also taught that ratios are commonly expressed as fractions, and that ratios can therefore be similarly reduced.

When the `<ratio>` header was first designed (circa 1999), we decided (a) to allow any representable ratio (whether reduced or unreduced) to be expressed, and (b) to provide a type alias member and corresponding static data members denoting the ratio's reduced equivalent. This design resulted in the specification that is today found in subclause [ratio.ratio] of the post-Kona Working Draft [N4659]:

```
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>;
  };
}
```

1 If the template argument **D** is zero or the absolute values of either of the template arguments **N** and **D** is not representable by type **intmax_t**, the program is ill-formed. [*Note:* These rules ensure that infinite ratios are avoided and that for any negative input, there exists a representable value of its absolute value which is positive. In a two's complement representation, this excludes the most negative value. — *end note*]

2 The static data members **num** and **den** shall have the following values, where **gcd** represents the greatest common divisor of the absolute values of **N** and **D**:

(2.1) — **num** shall have the value **sign(N) * sign(D) * abs(N) / gcd**.

(2.2) — **den** shall have the value **abs(D) / gcd**.

In that C++98 era, this seemed the best we could do. Alas, these decisions regrettably allowed a potentially large number of types to denote the same notional value: for example, the ratio **1** : **2** could be represented by the types **ratio<2,4>**, **ratio<5,10>**, **ratio<1,2>**, or any of many, many others. Thus, **<ratio>** users must take into account that any particular **ratio** type may be in unreduced form. Unfortunately, this consequence is sometimes overlooked. Even when taken into account, it has led to convoluted and error-prone code.

Via aliases, however, modern C++ allows us to ensure that such currently-distinct types (**ratio<2,4>**, **ratio<5,10>**, **ratio<1,2>**, etc.) all denote a single type, namely the equivalent reduced type **ratio<1,2>**. As illustrated in the next section (and as had been promised in [P0548R0]), this paper **proposes to adjust `ratio`'s specification** so as to do just that, and thus obtain only reduced **ratio** types.

This change will simplify reasoning about types (such as **std::duration**) that rely on **<ratio>**. It may also somewhat simplify their implementation, as they no longer need concern themselves with unreduced **ratio** types. Moreover, it will simplify specification of ratio arithmetic, as shown in §3, by removing extra wording to require reduced results.

2 Expository implementation

In brief, our strategy here[1] is (a) to rename today's **ratio** as, say, **__ratio**, and then (b) to redefine **ratio<N,D>** as an alias template for the equivalent **__ratio<N,D>::type**. (We do not show the helpers **num_min_v**, **static_abs_v**, **static_gcd_v**, and **static_sign_v**, as we believe their intent is as obvious as their implementation is straightforward.)

```
1 template< intmax_t N, intmax_t D >
2 class __ratio  {  // same members as the former ratio template
3   static_assert( N != num_min_v<intmax_t> );
4   static_assert( D != 0 and D != num_min_v<intmax_t> );
```

[1] See §4 for an alternate approach.

```
private:
  static constexpr intmax_t  abs_N = static_abs_v<N>;
  static constexpr intmax_t  abs_D = static_abs_v<D>;
  static constexpr intmax_t  gcd   = static_gcd_v<abs_N, abs_D>;

public:
  // reduced ratio components
  static constexpr intmax_t  num = static_sign_v<D> * N / gcd;
  static constexpr intmax_t  den = abs_D / gcd;

  // reduced ratio
  using type = __ratio<num, den>;
};  // __ratio<,>

template< intmax_t N, intmax_t D = 1 >
using
  ratio = typename __ratio<N, D>::type;  // alias the reduced form
```

We validated the above updated implementation against the same public and private tests used to validate implementations of the present specification. Every test passed unmodified.[2]

3 Proposed wording[3]

3.1 Edit [ratio.ratio] (23.16.3) as shown:

```
namespace std {
  template <intmax_t N, intmax_t D = 1>
    class ratioRatio {  // exposition only
    public:
      static constexpr intmax_t num;  // see below
      static constexpr intmax_t den;  // see below
      using type = ratioRatio<num, den>;
  };
  template <intmax_t N, intmax_t D = 1>
    using ratio = typename Ratio<N, D>::type;
}
```

1 IfThe program is ill-formed if (a) the template argument **D** is zero or (b) the absolute values of either of the template arguments **N** and **D** is not representable by type **intmax_t**, the program is ill-formed. [*Note:* These rules ensure (a) that infinite ratios are avoided and (b) that for any negative input, there exists a representable value of its absolute value which is positive. In a two's complement representation, this excludes the most negative value. — *end note*]

2 The static data members **num** and **den** shall have the following values, where **gcd** represents the greatest common divisor of the absolute values of **N** and **D**:

(2.1) — **num** shall have the value **sign(N) * sign(D) * abs(N) / gcd**.

[2] There were no tests expecting **is_same<ratio<A,B>,ratio<C,D>>** to be **false** in some cases. Some expressions of this form would become **true** under the present proposal and thus would likely have failed such an academic, implementation-oriented test.

[3] All proposed additions and deletions are relative to the post-Kona Working Draft [N4659]. Editorial notes are displayed against a gray background.

(2.2) — **den** shall have the value **abs(D) / gcd**.

3 [*Example*: Each of the following expressions is **true**.

- **is_same_v<ratio<5,10>, ratio<1,2>>**
- **is_same_v<ratio<5,10>, ratio<2,4>>**
- **is_same_v<ratio<2,4>, ratio<1,2>>**

— *end example*]

3.2 Edit [ratio.arithmetic] (23.16.4) as shown:

1 Each of the alias templates **ratio_add**, **ratio_subtract**, **ratio_multiply**, and **ratio_divide** denotes the result of an arithmetic computation on two **ratios R1** and **R2**. With ~~X~~N and ~~Y~~D computed (in the absence of arithmetic overflow) as specified by Table 51, each alias denotes ~~a **ratio<U, V>** such that **U** is the same as **ratio<X, Y>::num** and **V** is the same as **ratio<X, Y>::den**~~**ratio<N, D>**.

2 If it is not possible to represent ~~U~~N or ~~V~~D with **intmax_t**, the program is ill-formed. Otherwise, an implementation should yield correct values of ~~U~~N and ~~V~~D. If it is not possible to represent ~~X or Y~~all intermediate results with **intmax_t**, the program is ill-formed unless the implementation nonetheless yields correct values of ~~U~~N and ~~V~~D.

3.3 Edit column headings of Table 51 as shown:

Type	**Value of ~~X~~N**	**Value of ~~Y~~D**

4 An alternative

We also considered a different approach that does not alter **ratio**'s specification, but introduces the following new alias family instead:

```
template< intmax_t N, intmax_t D = 1 >
using
  ratio_t = typename ratio<N, D>::type;  // reduced form of ratio<N,D>
```

A different name is, of course, possible; **reduced_ratio_t** has already been suggested.

While we do not object to introducing such a new *typedef-name*, it does require users to be explicit about when they want a **ratio** in its reduced form. Based on our test results from the first approach, we find no cognitive, technical, or computational benefit in maintaining any unreduced **ratio**, nor in requiring users to specify when a reduced **ratio** is wanted.

5 Acknowledgments

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

6 Bibliography

[N4659] Richard Smith: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N4659 (post-Kona mailing), 2017–03–21. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf.

[P0548R0] Walter E. Brown: "**`common_type`** and **`duration`**." ISO/IEC JTC1/SC22/WG21 document P0548R0 (pre-Kona mailing), 2017–02–01. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0548r0.pdf.

7 Document history

Rev.	Date	Changes
0	2017-06-11	• Published as P0656R0.