## D.6. <ratio> header

The <ratio> header provides support for compile-time rational arithmetic.

*Header contents*

```
namespace std
{
    template<intmax_t N,intmax_t D=1>
    class ratio;

    // ratio arithmetic
    template <class R1, class R2>
    using ratio_add = see description;

    template <class R1, class R2>
    using ratio_subtract = see description ;

    template <class R1, class R2>
    using ratio_multiply = see description ;

    template <class R1, class R2>
    using ratio_divide = see description;
    // ratio comparison
    template <class R1, class R2>
    struct ratio_equal;

    template <class R1, class R2>
    struct ratio_not_equal;

    template <class R1, class R2>
    struct ratio_less;

    template <class R1, class R2>
    struct ratio_less_equal;

    template <class R1, class R2>
    struct ratio_greater;

    template <class R1, class R2>
    struct ratio_greater_equal;

    typedef ratio<1, 1000000000000000000> atto;
    typedef ratio<1, 1000000000000000> femto;
    typedef ratio<1, 1000000000000> pico;
    typedef ratio<1, 1000000000> nano;
    typedef ratio<1, 1000000> micro;
    typedef ratio<1, 1000> milli;
    typedef ratio<1, 100> centi;
    typedef ratio<1, 10> deci;
    typedef ratio<10, 1> deca;
    typedef ratio<100, 1> hecto;
    typedef ratio<1000, 1> kilo;
    typedef ratio<1000000, 1> mega;
    typedef ratio<1000000000, 1> giga;
    typedef ratio<1000000000000, 1> tera;
    typedef ratio<1000000000000000, 1> peta;
    typedef ratio<1000000000000000000, 1> exa;
}
```

### D.6.1. std::ratio class template

The `std::ratio` class template provides a mechanism for compile-time arithmetic involving rational values such as one half (`std::ratio<1,2>`), two thirds (`std::ratio<2,3>`) or fifteen forty-thirds (`std::ratio<15,43>`). It's used within the C++ Standard Library for specifying the period for instantiating the `std::chrono::duration` class template.

*Class definition*

```
template <intmax_t N, intmax_t D = 1>
class ratio
{
public:
    typedef ratio<num, den> type;
    static constexpr intmax_t num= see below;
    static constexpr intmax_t den= see below;
};
```

*Requirements*

`D` may not be zero.

*Description*

`num` and `den` are the numerator and denominator of the fraction `N/D` reduced to lowest terms. `den` is always positive. If `N` and `D` are the same sign, `num` is positive; otherwise `num` is negative.

*Examples*

```
ratio<4,6>::num == 2
ratio<4,6>::den == 3
ratio<4,-6>::num == -2
ratio<4,-6>::den == 3
```

### D.6.2. std::ratio_add template alias

The `std::ratio_add` template alias provides a mechanism for adding two `std::ratio` values at compile time, using rational arithmetic.

*Definition*

```
template <class R1, class R2>
using ratio_add = std::ratio<see below>;
```

*Preconditions*

R1 and R2 must be instantiations of the std::ratio class template.

*Effects*

ratio_add<R1,R2> is defined as an alias for an instantiation of std::ratio that represents the sum of the fractions represented by R1 and R2 if that sum can be calculated without overflow. If the calculation of the result overflows, the program is ill formed. In the absence of arithmetic overflow, std::ratio_add<R1,R2> shall have the same num and den values as std::ratio<R1::num * R2::den + R2::num * R1::den, R1::den * R2::den>.

*Examples*

```
std::ratio_add<std::ratio<1,3>, std::ratio<2,5> >::num == 11
std::ratio_add<std::ratio<1,3>, std::ratio<2,5> >::den == 15

std::ratio_add<std::ratio<1,3>, std::ratio<7,6> >::num == 3
std::ratio_add<std::ratio<1,3>, std::ratio<7,6> >::den == 2
```

### D.6.3. std::ratio_subtract template alias

The std::ratio_subtract template alias provides a mechanism for subtracting two std::ratio values at compile time, using rational arithmetic.

*Definition*

```
template <class R1, class R2>
using ratio_subtract = std::ratio<see below>;
```

*Preconditions*

R1 and R2 must be instantiations of the std::ratio class template.

*Effects*

ratio_subtract<R1,R2> is defined as an alias for an instantiation of std::ratio that represents the difference of the fractions represented by R1 and R2 if that difference can be calculated without overflow. If the calculation of the result overflows, the program is ill formed. In the absence of arithmetic overflow, std::ratio_subtract<R1,R2> shall have the same num and den values as std::ratio<R1::num * R2::den - R2::num * R1::den, R1::den * R2::den>.

*Examples*

```
std::ratio_subtract<std::ratio<1,3>, std::ratio<1,5> >::num == 2
std::ratio_subtract<std::ratio<1,3>, std::ratio<1,5> >::den == 15

std::ratio_subtract<std::ratio<1,3>, std::ratio<7,6> >::num == -5
std::ratio_subtract<std::ratio<1,3>, std::ratio<7,6> >::den == 6
```

### D.6.4. std::ratio_multiply template alias

The std::ratio_multiply template alias provides a mechanism for multiplying two std::ratio values at compile time, using rational arithmetic.

*Definition*

```
template <class R1, class R2>
using ratio_multiply = std::ratio<see below>;
```

*Preconditions*

`R1` and `R2` must be instantiations of the `std::ratio` class template.

*Effects*

`ratio_multiply<R1,R2>` is defined as an alias for an instantiation of `std::ratio` that represents the product of the fractions represented by `R1` and `R2` if that product can be calculated without overflow. If the calculation of the result overflows, the program is ill formed. In the absence of arithmetic overflow, `std::ratio_multiply<R1,R2>` shall have the same `num` and `den` values as `std::ratio<R1::num * R2::num, R1::den * R2::den>`.

*Examples*

```
std::ratio_multiply<std::ratio<1,3>, std::ratio<2,5> >::num == 2
std::ratio_multiply<std::ratio<1,3>, std::ratio<2,5> >::den == 15

std::ratio_multiply<std::ratio<1,3>, std::ratio<15,7> >::num == 5
std::ratio_multiply<std::ratio<1,3>, std::ratio<15,7> >::den == 7
```

### D.6.5. std::ratio_divide template alias

The `std::ratio_divide` template alias provides a mechanism for dividing two `std::ratio` values at compile time, using rational arithmetic.

*Definition*

```
template <class R1, class R2>
using ratio_divide = std::ratio<see below>;
```

*Preconditions*

`R1` and `R2` must be instantiations of the `std::ratio` class template.

*Effects*

`ratio_divide<R1,R2>` is defined as an alias for an instantiation of `std::ratio` that represents the result of dividing the fractions represented by `R1` and `R2` if that result can be calculated without overflow. If the calculation overflows, the program is ill formed. In the absence of arithmetic overflow, `std::ratio_divide<R1,R2>` shall have the same `num` and `den` values as `std::ratio<R1::num * R2::den, R1::den *R2::num>`.

*Examples*

```
std::ratio_divide<std::ratio<1,3>, std::ratio<2,5> >::num == 5
std::ratio_divide<std::ratio<1,3>, std::ratio<2,5> >::den == 6

std::ratio_divide<std::ratio<1,3>, std::ratio<15,7> >::num == 7
```

```
std::ratio_divide<std::ratio<1,3>, std::ratio<15,7> >::den == 45
```

### D.6.6. std::ratio_equal class template

The `std::ratio_equal` class template provides a mechanism for comparing two `std::ratio` values for equality at compile time, using rational arithmetic.

*Class definition*

```
template <class R1, class R2>
class ratio_equal:
    public std::integral_constant<
        bool,(R1::num == R2::num) && (R1::den == R2::den)>
{};
```

*Preconditions*

`R1` and `R2` must be instantiations of the `std::ratio` class template.

*Examples*

```
std::ratio_equal<std::ratio<1,3>, std::ratio<2,6> >::value == true
std::ratio_equal<std::ratio<1,3>, std::ratio<1,6> >::value == false
std::ratio_equal<std::ratio<1,3>, std::ratio<2,3> >::value == false
std::ratio_equal<std::ratio<1,3>, std::ratio<1,3> >::value == true
```

### D.6.7. std::ratio_not_equal class template

The `std::ratio_not_equal` class template provides a mechanism for comparing two `std::ratio` values for inequality at compile time, using rational arithmetic.

*Class definition*

```
template <class R1, class R2>
class ratio_not_equal:
    public std::integral_constant<bool, !ratio_equal<R1,R2>::value>
{};
```

*Preconditions*

`R1` and `R2` must be instantiations of the `std::ratio` class template.

*Examples*

```
std::ratio_not_equal<std::ratio<1,3>, std::ratio<2,6> >::value == false
std::ratio_not_equal<std::ratio<1,3>, std::ratio<1,6> >::value == true
std::ratio_not_equal<std::ratio<1,3>, std::ratio<2,3> >::value == true
std::ratio_not_equal<std::ratio<1,3>, std::ratio<1,3> >::value == false
```

### D.6.8. Std::Ratio_Less Class Template

The `std::ratio_less` class template provides a mechanism for comparing two `std::ratio` values at compile time, using rational arithmetic.

*Class definition*

```
template <class R1, class R2>
class ratio_less:
    public std::integral_constant<bool,see below>
{};
```

*Preconditions*

R1 and R2 must be instantiations of the std::ratio class template.

*Effects*

std::ratio_less<R1,R2> derives from std::integral_constant<bool, *value* >, where *value* is (R1::num*R2::den) < (R2::num*R1::den). Where possible, implementations shall use a method of calculating the result that avoids overflow. If overflow occurs, the program is ill formed.

*Examples*

```
std::ratio_less<std::ratio<1,3>, std::ratio<2,6> >::value == false
std::ratio_less<std::ratio<1,6>, std::ratio<1,3> >::value == true
std::ratio_less<
     std::ratio<999999999,1000000000>,
     std::ratio<1000000001,1000000000> >::value == true
std::ratio_less<
     std::ratio<1000000001,1000000000>,
     std::ratio<999999999,1000000000> >::value == false
```

### D.6.9. std::ratio_greater class template

The std::ratio_greater class template provides a mechanism for comparing two std::ratio values at compile time, using rational arithmetic.

*Class definition*

```
template <class R1, class R2>
class ratio_greater:
    public std::integral_constant<bool,ratio_less<R2,R1>::value>
{};
```

*Preconditions*

R1 and R2 must be instantiations of the std::ratio class template.

### D.6.10. std::ratio_less_equal class template

The std::ratio_less_equal class template provides a mechanism for comparing two std::ratio values at compile time, using rational arithmetic.

*Class definition*

```
template <class R1, class R2>
class ratio_less_equal:
    public std::integral_constant<bool,!ratio_less<R2,R1>::value>
{};
```

*Preconditions*

R1 and R2 must be instantiations of the std::ratio class template.

### D.6.11. std::ratio_greater_equal class template

The std::ratio_greater_equal class template provides a mechanism for comparing two std::ratio values at compile time, using rational arithmetic.

*Class definition*

```
template <class R1, class R2>
class ratio_greater_equal:
    public std::integral_constant<bool,!ratio_less<R1,R2>::value>
{};
```

*Preconditions*

R1 and R2 must be instantiations of the std::ratio class template.