Boost.Ratio 1.0.1

Howard Hinnant
Beman Dawes
Vicente J. Botet Escriba

Table of Contents

Overview

How to Use This Documentation

This documentation makes use of the following naming and formatting conventions.

- Code is in `fixed width font` and is syntax-highlighted.
- Replaceable text that you will need to supply is in *`italics`* .
- Free functions are rendered in the code font followed by `()`, as in `free_function()`.
- If a name refers to a class template, it is specified like this: `class_template<>`; that is, it is in code font and its name is followed by `<>` to indicate that it is a class template.
- If a name refers to a function-like macro, it is specified like this: `MACRO()`; that is, it is uppercase in code font and its name is followed by `()` to indicate that it is a function-like macro. Object-like macros appear without the trailing `()`.
- Names that refer to *concepts* in the generic programming sense are specified in CamelCase.

> **Note**
>
> In addition, notes such as this one specify non-essential information that provides additional background or rationale.

Finally, you can mentally add the following to any code fragments in this document:

```
    // Include all of Ratio files
    #include <boost/ratio.hpp>
    using namespace boost;
sec
```

Motivation

Boost.Ratio aims to implement the compile time ratio facility in C++0x, as proposed in **N2661 - A Foundation to Sleep On**. That document provides background and motivation for key design decisions and is the source of a good deal of information in this documentation.

Description

The **Boost.Ratio** library provides:

- A class template, `ratio`, for specifying compile time rational constants such as 1/3 of a nanosecond or the number of inches per meter. `ratio` represents a compile time ratio of compile time constants with support for compile time arithmetic with overflow and division by zero protection.
- It provides a textual representation of `boost::ratio<N, D>` in the form of a `std::basic_string` which can be useful for I/O.
- Some extension related to the Rational Constant concept enabling the use of `ratio`<> in the context of **Boost.MPL** numeric metafunctions.

User's Guide

Getting Started

Installing Ratio

Getting Boost.Ratio

Boost.Ratio is in the latest Boost release in the folder `/boost/ratio`. Documentation, tests and examples folder are at `boost/libs/ratio/`.

You can also access the latest (unstable?) state from the Boost trunk directories `boost/ratio` and `libs/ratio`.

Just go to the wiki and follow the instructions there for anonymous SVN access.

Where to install Boost.Ratio?

The simple way is to decompress (or checkout from SVN) the files in your BOOST_ROOT directory.

Building Boost.Ratio

Boost.Ratio is a header only library, so no need to compile anything, you just need to `include <boost/ratio.hpp>`.

Requirements

Boost.Ratio depends on some Boost libraries. For these specific parts you must use either Boost version 1.39.0 or later (even older versions may work).

In particular, **Boost.Ratio** depends on:

Boost.Config	for configuration purposes, ...
Boost.Integer	for cstdint conformance, and integer traits ...
Boost.MPL	for MPL Assert and bool, logical ...
Boost.StaticAssert	for STATIC_ASSERT, ...
Boost.TypeTraits	for is_base, is_convertible ...
Boost.Utility/EnableIf	for enable_if, ...

Building an executable that uses Boost.Ratio

No link is needed.

Exception safety

All functions in the library are exception-neutral, providing the strong exception safety guarantee.

Thread safety

All functions in the library are thread-unsafe except when noted explicitly.

Tested compilers

Boost.Ratio should work with an C++03 conforming compiler. The current version has been tested on:

Windows with

- MSVC 10.0
- MSVC 9.0 Express
- MSVC 8.0

Cygwin 1.5 with

- GCC 3.4.4

Cygwin 1.7 with

- GCC 4.3.4

MinGW with

- GCC 4.4.0
- GCC 4.5.0
- GCC 4.5.0 -std=c++0x
- GCC 4.6.0
- GCC 4.6.0 -std=c++0x

Initial versions were tested on:

MacOS with GCC 4.2.4

Ubuntu Linux with GCC 4.2.4

Note

Please let us know how this works on other platforms/compilers.

Note

Please send any questions, comments and bug reports to boost <at> lists <dot> boost <dot> org.

Tutorial

Ratio

`ratio` is a general purpose utility inspired by Walter Brown allowing one to easily and safely compute rational values at compile-time. The `ratio` class catches all errors (such as divide by zero and overflow) at compile time. It is used in the duration and `time_point` classes to efficiently create units of time. It can also be used in other "quantity" libraries or anywhere there is a rational constant which is known at compile-time. The use of this utility can greatly reduce the chances of run-time overflow because the `ratio` (and any ratios resulting from `ratio` arithmetic) are always reduced to the lowest terms.

`ratio` is a template taking two `intmax_t`s, with the second defaulted to 1. In addition to copy constructors and assignment, it only has two public members, both of which are `static const intmax_t`. One is the numerator of the `ratio` and the other is the denominator. The `ratio` is always normalized such that it is expressed in lowest terms, and the denominator is always positive. When the numerator is 0, the denominator is always 1.

Example:

```
    typedef ratio<5, 3>   five_thirds;
    // five_thirds::num == 5, five_thirds::den == 3

    typedef ratio<25, 15> also_five_thirds;
    // also_five_thirds::num == 5, also_five_thirds::den == 3

    typedef ratio_divide<five_thirds, also_five_thirds>::type one;
    // one::num == 1, one::den == 1
asis> nu
```

This facility also includes convenience typedefs for the SI prefixes `atto` through `exa` corresponding to their internationally recognized definitions (in terms of `ratio`). This is a tremendous syntactic convenience. It will prevent errors in specifying constants as one no longer has to double count the number of zeros when trying to write millions or billions.

Example:

```
    typedef ratio_multiply<ratio<5>, giga>::type _5giga;
    // _5giga::num == 5000000000, _5giga::den == 1

    typedef ratio_multiply<ratio<5>, nano>::type _5nano;
    // _5nano::num == 1, _5nano::den == 200000000
def r
```

Ratio I/O

For each `ratio<N, D>` there exists a `ratio_string<ratio<N, D>, CharT>` for which you can query two strings: `short_name` and `long_name`. For those `ratio`'s that correspond to an SI prefix long_name corresponds to the internationally recognized prefix, stored as a `basic_string<CharT>`. For example `ratio_string<mega, char>::long_name()` returns `string("mega")`. For those `ratio`s that correspond to an SI prefix `short_name` corresponds to the internationally recognized symbol, stored as a `basic_string<CharT>`. For example, `ratio_string<mega, char>::short_name()` returns `string("M")`. For all other `ratio`s, both `long_name()` and `short_name()` return a `basic_string` containing "[`ratio::num/ratio::den`]".

`ratio_string<ratio<N, D>, CharT>` is only defined for four character types:

- `char`: UTF-8
- `char16_t`: UTF-16
- `char32_t`: UTF-32
- `wchar_t`: UTF-16 (if wchar_t is 16 bits) or UTF-32

When the character is char, UTF-8 will be used to encode the names. When the character is `char16_t`, UTF-16 will be used to encode the names. When the character is `char32_t`, UTF-32 will be used to encode the names. When the character is `wchar_t`, the encoding will be UTF-16 if `wchar_t` is 16 bits, and otherwise UTF-32.

The `short_name` (Greek mu or μ) for micro is defined by Unicode to be U+00B5.

Examples:

```
#include <boost/ratio/ratio_io.hpp>
#include <iostream>

int main()
{
    using namespace std;
    using namespace boost;

    cout << "ratio_string<deca, char>::long_name() = "
         <<  ratio_string<deca, char>::long_name() << '\n';
    cout << "ratio_string<deca, char>::short_name() = "
         <<  ratio_string<deca, char>::short_name() << '\n';

    cout << "ratio_string<giga, char>::long_name() = "
         <<  ratio_string<giga, char>::long_name() << '\n';
    cout << "ratio_string<giga, char>::short_name() = "
         <<  ratio_string<giga, char>::short_name() << '\n';

    cout << "ratio_string<ratio<4, 6>, char>::long_name() = "
         <<  ratio_string<ratio<4, 6>, char>::long_name() << '\n';
    cout << "ratio_string<ratio<4, 6>, char>::short_name() = "
         <<  ratio_string<ratio<4, 6>, char>::short_name() << '\n';
}
.org/wiki/SI_prefix#Lis
```

The output will be

```
    ratio_string<deca, char>::long_name() = deca
    ratio_string<deca, char>::short_name() = da
    ratio_string<giga, char>::long_name() = giga
    ratio_string<giga, char>::short_name() = G
    ratio_string<ratio<4, 6>, char>::long_name() = [2/3]
    ratio_string<ratio<4, 6>, char>::short_name() = [2/3]
ntifie
```

Ratio MPL Numeric Metafunctions

With the view of the _ratio class as a Rational Constant we can mix _ratio<> and **Boost.MPL** Integral Constants in the same expression, as in

```
    typedef mpl::times<int_<5>, giga>::type _5giga;
    // _5giga::num == 5000000000, _5giga::den == 1

    typedef mpl::times<int_<5>, nano>::type _5nano;
    // _5nano::num == 1, _5nano::den == 200000000
s####
```

Example

SI units

This example illustrates the use of type-safe physics code interoperating with `boost::chrono::duration` types, taking advantage of the **Boost.Ratio** infrastructure and design philosophy.

Let's start by defining a `length` class template that mimics `boost::chrono::duration`, which represents a time duration in various units, but restricts the representation to `double` and uses **Boost.Ratio** for length unit conversions:

```
    template <class Ratio>
    class length {
    private:
        double len_;
    public:
        typedef Ratio ratio;
        length() : len_(1) {}
        length(const double& len) : len_(len) {}

        template <class R>
        length(const length<R>& d)
                : len_(d.count() * boost::ratio_divide<Ratio, R>::type::den /
                                   boost::ratio_divide<Ratio, R>::type::num) {}

        double count() const {return len_;}

        length& operator+=(const length& d) {len_ += d.count(); return *this;}
        length& operator-=(const length& d) {len_ -= d.count(); return *this;}

        length operator+() const {return *this;}
        length operator-() const {return length(-len_);}

        length& operator*=(double rhs) {len_ *= rhs; return *this;}
        length& operator/=(double rhs) {len_ /= rhs; return *this;}
    };
ng<ratio<4, 6>, char>::sh
```

Here's a small sampling of length units:

```
    typedef length<boost::ratio<1> >          meter;        // set meter as "unity"
    typedef length<boost::centi>              centimeter;   // 1/100 meter
    typedef length<boost::kilo>               kilometer;    // 1000  meters
    typedef length<boost::ratio<254, 10000> > inch;         // 254/10000 meters
umer
```

Note that since `length`'s template parameter is actually a generic ratio type, so we can use boost::ratio allowing for more complex length units:

```
    typedef length<boost::ratio_multiply<boost::ratio<12>, inch::ratio>::type>   foot;  // 12 inchs
    typedef length<boost::ratio_multiply<boost::ratio<5280>, foot::ratio>::type> mile;  // 5280 ↵
feet
##
```

Now we need a floating point-based definition of seconds:

```
    typedef boost::chrono::duration<double> seconds;                         // unity
#
```

We can even support sub-nanosecond durations:

```
    typedef boost::chrono::duration<double,  boost::pico> picosecond;  // 10^-12 seconds
    typedef boost::chrono::duration<double, boost::femto> femtosecond; // 10^-15 seconds
    typedef boost::chrono::duration<double,  boost::atto> attosecond;  // 10^-18 seconds
o<2
```

Finally, we can write a proof-of-concept of an SI units library, hard-wired for meters and floating point seconds, though it will accept other units:

```
    template <class R1, class R2>
    class quantity
    {
        double q_;
    public:
        typedef R1 time_dim;
        typedef R2 distance_dim;
        quantity() : q_(1) {}

        double get() const {return q_;}
        void set(double q) {q_ = q;}
    };

    template <>
    class quantity<boost::ratio<1>, boost::ratio<0> >
    {
        double q_;
    public:
        quantity() : q_(1) {}
        quantity(seconds d) : q_(d.count()) {}  // note:  only User1::seconds needed here

        double get() const {return q_;}
        void set(double q) {q_ = q;}
    };

    template <>
    class quantity<boost::ratio<0>, boost::ratio<1> >
    {
        double q_;
    public:
        quantity() : q_(1) {}
        quantity(meter d) : q_(d.count()) {}  // note:  only User1::meter needed here

        double get() const {return q_;}
        void set(double q) {q_ = q;}
    };

    template <>
    class quantity<boost::ratio<0>, boost::ratio<0> >
    {
        double q_;
    public:
        quantity() : q_(1) {}
        quantity(double d) : q_(d) {}

        double get() const {return q_;}
        void set(double q) {q_ = q;}
    };
e="identifier">CharT</phrase><phrase role=#speci
```

That allows us to create some useful SI-based unit types:

```
    typedef quantity<boost::ratio<0>, boost::ratio<0> >  Scalar;
    typedef quantity<boost::ratio<1>, boost::ratio<0> >  Time;         // second
    typedef quantity<boost::ratio<0>, boost::ratio<1> >  Distance;     // meter
    typedef quantity<boost::ratio<-1>, boost::ratio<1> > Speed;        // meter/second
    typedef quantity<boost::ratio<-2>, boost::ratio<1> > Acceleration; // meter/second^2
ase r
```

To make quantity useful, we need to be able to do arithmetic:

```
    template <class R1, class R2, class R3, class R4>
    quantity<typename boost::ratio_subtract<R1, R3>::type,
             typename boost::ratio_subtract<R2, R4>::type>
    operator/(const quantity<R1, R2>& x, const quantity<R3, R4>& y)
    {
        typedef quantity<typename boost::ratio_subtract<R1, R3>::type,
                         typename boost::ratio_subtract<R2, R4>::type> R;
        R r;
        r.set(x.get() / y.get());
        return r;
    }

    template <class R1, class R2, class R3, class R4>
    quantity<typename boost::ratio_add<R1, R3>::type,
             typename boost::ratio_add<R2, R4>::type>
    operator*(const quantity<R1, R2>& x, const quantity<R3, R4>& y)
    {
        typedef quantity<typename boost::ratio_add<R1, R3>::type,
                         typename boost::ratio_add<R2, R4>::type> R;
        R r;
        r.set(x.get() * y.get());
        return r;
    }

    template <class R1, class R2>
    quantity<R1, R2>
    operator+(const quantity<R1, R2>& x, const quantity<R1, R2>& y)
    {
        typedef quantity<R1, R2> R;
        R r;
        r.set(x.get() + y.get());
        return r;
    }

    template <class R1, class R2>
    quantity<R1, R2>
    operator-(const quantity<R1, R2>& x, const quantity<R1, R2>& y)
    {
        typedef quantity<R1, R2> R;
        R r;
        r.set(x.get() - y.get());
        return r;
    }
tio"><code><phrase role="identifier#>ratio<
```

With all of the foregoing scaffolding, we can now write an exemplar of a type-safe physics function:

```
    Distance
    compute_distance(Speed v0, Time t, Acceleration a)
    {
        return v0 * t + Scalar(.5) * a * t * t;  // if a units mistake is made here it won't com↵
pile
    }
ngth ↵
```

Finally, we can exercise what we've created, even using custom time durations (`User1::seconds`) as well as Boost time durations (`boost::chrono::hours`). The input can be in arbitrary, though type-safe, units, the output is always in SI units. (A complete Units library would support other units, of course.)

```
    int main()
    {
        typedef boost::ratio<8, BOOST_INTMAX_C(0x7FFFFFFD)> R1;
        typedef boost::ratio<3, BOOST_INTMAX_C(0x7FFFFFFD)> R2;
        typedef User1::quantity<boost::ratio_subtract<boost::ratio<0>, boost::ratio<1> >::type,
                                 boost::ratio_subtract<boost::ratio<1>, boost::ra↲
tio<0> >::type > RR;
        typedef boost::ratio_subtract<R1, R2>::type RS;
        std::cout << RS::num << '/' << RS::den << '\n';

        std::cout << "*************\n";
        std::cout << "* testUser1 *\n";
        std::cout << "*************\n";
        User1::Distance d( User1::mile(110) );
        User1::Time t( boost::chrono::hours(2) );

        RR r=d / t;
        //r.set(d.get() / t.get());

        User1::Speed rc= r;

        User1::Speed s = d / t;
        std::cout << "Speed = " << s.get() << " meters/sec\n";
        User1::Acceleration a = User1::Dis↲
tance( User1::foot(32.2) ) / User1::Time() / User1::Time();
        std::cout << "Acceleration = " << a.get() << " meters/sec^2\n";
        User1::Distance df = compute_distance(s, User1::Time( User1::seconds(0.5) ), a);
        std::cout << "Distance = " << df.get() << " meters\n";
        std::cout << "There are "
            << User1::mile::ratio::den << '/' << User1::mile::ratio::num << " miles/meter";
        User1::meter mt = 1;
        User1::mile mi = mt;
        std::cout << " which is approximately " << mi.count() << '\n';
        std::cout << "There are "
            << User1::mile::ratio::num << '/' << User1::mile::ratio::den << " meters/mile";
        mi = 1;
        mt = mi;
        std::cout << " which is approximately " << mt.count() << '\n';
        User1::attosecond as(1);
        User1::seconds sec = as;
        std::cout << "1 attosecond is " << sec.count() << " seconds\n";
        std::cout << "sec = as;  // compiles\n";
        sec = User1::seconds(1);
        as = sec;
        std::cout << "1 second is " << as.count() << " attoseconds\n";
        std::cout << "as = sec;  // compiles\n";
        std::cout << "\n";
        return 0;
    }
se role="special">&gt;,</phrase> <phrase role=#i
```

See the source file example/si_physics.cpp

External Resources

C++ Standards Committee's current Working Paper	The most authoritative reference material for the library is the C++ Standards Committee's current Working Paper (WP). 20.6 Compile-time rational arithmetic "ratio"
N2661 - A Foundation to Sleep On	From Howard E. Hinnant, Walter E. Brown, Jeff Garland and Marc Paterno. Is very informative and provides motivation for key design decisions

LWG 1281. CopyConstruction and Assignment between ratios having the same normalized form From Vicente Juan Botet Escriba.

Reference

C++0x Recommendation

Header `<boost/ratio.hpp>`

This header includes all the ratio related header files

```
    #include <boost/ratio/ratio.hpp>
    #include <boost/ratio/ratio_io.hpp>
    #include <boost/ratio/rational_constant.hpp>
.</
```

Header `<boost/ratio/ratio_fwd.hpp>`

This header provides forward declarations for the `<boost/ratio/ratio.hpp>` file.

```
   namespace boost  {

       template <boost::intmax_t N, boost::intmax_t D = 1> class ratio;

       // ratio arithmetic
       template <class R1, class R2> struct ratio_add;
       template <class R1, class R2> struct ratio_subtract;
       template <class R1, class R2> struct ratio_multiply;
       template <class R1, class R2> struct ratio_divide;
       template <class R> struct ratio_negate;
       template <class R> struct ratio_sign;
       template <class R> struct ratio_abs;
       template <class R1, class R2> struct ratio_gcd;
       template <class R1, class R2> struct ratio_lcm;

       // 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;

       // convenience SI typedefs
       typedef ratio<1LL, 1000000000000000000LL> atto;
       typedef ratio<1LL,    1000000000000000LL> femto;
       typedef ratio<1LL,       1000000000000LL> pico;
       typedef ratio<1LL,          1000000000LL> nano;
       typedef ratio<1LL,             1000000LL> micro;
       typedef ratio<1LL,                1000LL> milli;
       typedef ratio<1LL,                 100LL> centi;
       typedef ratio<1LL,                  10LL> deci;
       typedef ratio<                 10LL, 1LL> deca;
       typedef ratio<                100LL, 1LL> hecto;
       typedef ratio<               1000LL, 1LL> kilo;
       typedef ratio<            1000000LL, 1LL> mega;
       typedef ratio<         1000000000LL, 1LL> giga;
       typedef ratio<      1000000000000LL, 1LL> tera;
       typedef ratio<   1000000000000000LL, 1LL> peta;
       typedef ratio<1000000000000000000LL, 1LL> exa;
   }
<< "as = sec;  // compiles\n";
       s
```

Header `<boost/ratio/ratio.hpp>`

`ratio` is a facility which is useful in specifying compile-time rational constants. Compile-time rational arithmetic is supported with protection against overflow and divide by zero. Such a facility is very handy to efficiently represent 1/3 of a nanosecond, or to specify an inch in terms of meters (for example 254/10000 meters - which `ratio` will reduce to 127/5000 meters).

```
    // Configuration macros
    #define BOOST_RATIO_USES_STATIC_ASSERT
    #define BOOST_RATIO_USES_MPL_ASSERT
    #define BOOST_RATIO_USES_ARRAY_ASSERT
    #define BOOST_RATIO_EXTENSIONS
rase>
```

Configuration Macros

When BOOST_NO_STATIC_ASSERT is defined, the user can select the way static assertions are reported. Define

- BOOST_RATIO_USES_STATIC_ASSERT to use Boost.StaticAssert.
- BOOST_RATIO_USES_MPL_ASSERT to use **Boost.MPL** static assertions.
- BOOST_RATIO_USES_RATIO_ASSERT to use **Boost.Ratio** static assertions.

The default behavior is as if BOOST_RATIO_USES_ARRAY_ASSERT is defined.

When BOOST_RATIO_USES_MPL_ASSERT is not defined the following symbols are defined as shown:

```
    #define BOOST_RATIO_OVERFLOW_IN_ADD "overflow in ratio add"
    #define BOOST_RATIO_OVERFLOW_IN_SUB "overflow in ratio sub"
    #define BOOST_RATIO_OVERFLOW_IN_MUL "overflow in ratio mul"
    #define BOOST_RATIO_OVERFLOW_IN_DIV "overflow in ratio div"
    #define BOOST_RATIO_NUMERATOR_IS_OUT_OF_RANGE "ratio numerator is out of range"
    #define BOOST_RATIO_DIVIDE_BY_0 "ratio divide by 0"
    #define BOOST_RATIO_DENOMINATOR_IS_OUT_OF_RANGE "ratio denominator is out of range"
rase> #
```

Depending upon the static assertion system used, a hint as to the failing assertion will appear in some form in the compiler diagnostic output.

When BOOST_RATIO_EXTENSIONS is defined, **Boost.Ratio** provides in addition some extenion to the C++ standard, see below.

Class Template `ratio<>`

```
        template <boost::intmax_t N, boost::intmax_t D>
        class ratio {
        public:
            static const boost::intmax_t num;
            static const boost::intmax_t den;
            typedef ratio<num, den> type;

            #ifdef BOOST_RATIO_EXTENSIONS
            typedef mpl::rational_c_tag tag;
            typedef boost::rational<boost::intmax_t> value_type;
            typedef boost::intmax_t num_type;
            typedef boost::intmax_t den_type;

            ratio() = default;

            template <intmax_t _N2, intmax_t _D2>
            ratio(const ratio<_N2, _D2>&);

            template <intmax_t _N2, intmax_t _D2>
            ratio& operator=(const ratio<_N2, _D2>&);

            static value_type value();
            value_type operator()() const;
            #endif
        };
ase><phrase role=#identif
```

A diagnostic will be emitted if `ratio` is instantiated with `D == 0`, or if the absolute value of `N` or `D` cannot be represented. **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.

The members num and den will be normalized values of the template arguments N and D computed as follows. Let `gcd` denote the greatest common divisor of `N`'s absolute value and of `D`'s absolute value. Then:

- `num` has the value `sign(N)*sign(D)*abs(N)/gcd`.
- `den` has the value `abs(D)/gcd`.

The nested typedef `type` denotes the normalized form of this `ratio` type. It should be used when the normalized form of the template arguments are required, since the arguments are not necessarily normalized.

Two `ratio` classes `ratio<N1,D1>` and `ratio<N2,D2>` have the same normalized form if `ratio<N1,D1>::type` is the same type as `ratio<N2,D2>::type`

Construction and Assignment

Included only if BOOST_RATIO_EXTENSIONS is defined.

Default Constructor

```
      ratio()=default;
t
```

Effects: Constructs a `ratio` object.

Copy Constructor

```
    template <intmax_t N2, intmax_t D2>
      ratio(const ratio<N2, D2>& r);
>#
```

Effects: Constructs a `ratio` object.

Remarks: This constructor will not participate in overload resolution unless `r` has the same normalized form as `*this`.

Assignement

```
    template <intmax_t N2, intmax_t D2>
      ratio& operator=(const ratio<N2, D2>& r);
p#
```

Effects: Assigns a `ratio` object.

Returns: *this.

Remarks: This operator will not participate in overload resolution unless `r` has the same normalized form as `*this`.

MPL Numeric Metafunctions

Included only if BOOST_RATIO_EXTENSIONS is defined.

In order to work with **Boost.MPL** numeric metafunctions as a Rational Constant, the following has beed added:

```
            typedef mpl::rational_c_tag tag;
            typedef boost::rational<boost::intmax_t> value_type;
            typedef boost::intmax_t num_type;
            typedef boost::intmax_t den_type;
o</p
```

typedef mpl::rational_c_tag tag; typedef boost::rational<boost::intmax_t> value_type; typedef boost::intmax_t num_type; typedef boost::intmax_t den_type;

Observers

Included only if BOOST_RATIO_EXTENSIONS is defined.

```
            static value_type value();
            value_type operator()() const;
io
```

Returns: value_type(num,den);

`ratio` Arithmetic

For each of the class templates in this section, each template parameter refers to a `ratio`. If the implementation is unable to form the indicated `ratio` due to overflow, a diagnostic will be issued.

`ratio_add<>`

```
       template <class R1, class R2> struct ratio_add {
          typedef [/see below] type;
       };
##
```

The nested typedef `type` is a synonym for `ratio<R1::num * R2::den + R2::num * R1::den, R1::den * R2::den>::type`.

`ratio_subtract<>`

```
        template <class R1, class R2> struct ratio_subtract {
           typedef  [/see below]  type;
        };
lin
```

The nested typedef `type` is a synonym for `ratio<R1::num * R2::den - R2::num * R1::den, R1::den * R2::den>::type`.

`ratio_multiply<>`

```
        template <class R1, class R2> struct ratio_multiply {
           typedef  [/see below]  type;
        };
lin
```

The nested typedef `type` is a synonym for `ratio<R1::num * R2::num, R1::den * R2::den>::type`.

ratio_divide<>

```
        template <class R1, class R2> struct ratio_divide {
           typedef  [/see below]  type;
        };
a##
```

The nested typedef `type` is a synonym for `ratio<R1::num * R2::den, R2::num * R1::den>::type`.

ratio_negate<>

This extension of the C++ standard helps in the definition of some **Boost.MPL** numeric metafunctions.

```
        template <class R> struct ratio_negate {
           typedef  [/see below]  type;
        };
#>#
```

The nested typedef `type` is a synonym for `ratio<-R::num, R::den>::type`.

ratio_abs<>

This extension of the C++ standard helps in the definition of some **Boost.MPL** numeric metafunctions.

```
        template <class R> struct ratio_abs {
           typedef  [/see below]  type;
        };
L#
```

The nested typedef `type` is a synonym for `ratio<abs_c<intmax_t,R::num>::value, R::den>::type`.

ratio_sign<>

This extension of the C++ standard helps in the definition of some **Boost.MPL** numeric metafunctions.

```
        template <class R> struct ratio_sign {
           typedef  [/see below]  type;
        };
#
#
```

The nested typedef `type` is a synonym for `sign_c<intmax_t,R::num>::type`.

ratio_gcd<>

This extension of the C++ standard helps in the definition of some **Boost.MPL** numeric metafunctions.

```
        template <class R1, class R2> struct ratio_gcd {
           typedef  [/see below]  type;
        };
###
```

The nested typedef `type` is a synonym for `ratio<gcd_c<intmax_t, R1::num, R2::num>::value, mpl::lcm_c<intmax_t, R1::den, R2::den>::value>::type`.

`ratio_lcm<>`

This extension of the C++ standard helps in the definition of some **Boost.MPL** numeric metafunctions.

```
        template <class R1, class R2> struct ratio_lcm {
           typedef  [/see below]  type;
        };
###
```

The nested typedef `type` is a synonym for `ratio<lcm_c<intmax_t, R1::num, R2::num>::value, gcd_c<intmax_t, R1::den, R2::den>::value>::type`.

`ratio` Comparison

`ratio_equal<>`

```
        template <class R1, class R2>  struct ratio_equal
            : public boost::integral_constant<bool, [/see below] > {};
ns
```

If R1::num = `R2::num && R1::den` = R2::den, ratio_equal derives from true_type, else derives from false_type.

`ratio_not_equal<>`

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

`ratio_less<>`

```
        template <class R1, class R2>
        struct ratio_less
            : public boost::integral_constant<bool, [/see below] > {};
{};
```

If R1::num * R2::den < R2::num * R1::den, ratio_less derives from true_type, else derives from false_type.

`ratio_less_equal<>`

```
    template <class R1, class R2> struct ratio_less_equal
        : public boost::integral_constant<bool, !ratio_less<R2, R1>::value> {};
#h
```

`ratio_greater<>`

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

ratio_greater_equal<>

```
    template <class R1, class R2> struct ratio_greater_equal
        : public boost::integral_constant<bool, !ratio_less<R1, R2>::value> {};
#s
```

SI typedefs

The International System of Units specifies twenty SI prefixes. **Boost.Ratio** defines all except `yocto`, `zepto`, `zetta`, and `yotta`

```
        // convenience SI typedefs
        typedef ratio<1LL, 1000000000000000000LL> atto;
        typedef ratio<1LL,    1000000000000000LL> femto;
        typedef ratio<1LL,       1000000000000LL> pico;
        typedef ratio<1LL,          1000000000LL> nano;
        typedef ratio<1LL,             1000000LL> micro;
        typedef ratio<1LL,                1000LL> milli;
        typedef ratio<1LL,                 100LL> centi;
        typedef ratio<1LL,                  10LL> deci;
        typedef ratio<                 10LL, 1LL> deca;
        typedef ratio<                100LL, 1LL> hecto;
        typedef ratio<               1000LL, 1LL> kilo;
        typedef ratio<            1000000LL, 1LL> mega;
        typedef ratio<         1000000000LL, 1LL> giga;
        typedef ratio<      1000000000000LL, 1LL> tera;
        typedef ratio<   1000000000000000LL, 1LL> peta;
        typedef ratio<1000000000000000000LL, 1LL> exa;
e><phrase role=#i
```

Limitations

The following are limitations of Boost.Ratio relative to the specification in the C++0x draft standard:

- Four of the SI units typedefs -- `yocto`, `zepto`, `zetta`, and `yotta` -- are to be conditionally supported, if the range of `intmax_t` allows, but are not supported by **Boost.Ratio**.
- Ratio values should be of type static `constexpr intmax_t` (see Ratio values should be constexpr), but for compiler not supporting `constexpr` today, **Boost.Ratio** uses `static const intmax_t` instead.
- Rational arithmetic should use template aliases (see Rational Arithmetic should use template aliases), but those are not available in C++03, so inheritance is used instead.

Extensions

When `BOOST_RATIO_EXTENSIONS` is defined **Boost.Ratio** provides the following extensions:

- Extends the requirements of the C++0x draft standard by making the copy constructor and copy assignment operator have the same normalized form (see copy constructor and assignment between ratios having the same normalized form).
- More C++ standard like metafunctions applied to ratio types, like __static_abs or __static_negate.
- An __Boost_Mpl rational constant concept and the associated __Boost_Mpl arithmetic and comparison specializations including __numeric_cast, __plus, __equal_to between others.

Ratio I/O

Header `<boost/ratio/ratio_io.hpp>`

This header provides `ratio_string<>` which can generate a textual representation of a `ratio<>` in the form of a `std::basic_string<>`. These strings can be useful for I/O.

```
    namespace boost {
        template <class Ratio, class CharT>
        struct ratio_string
        {
            static std::basic_string<CharT> short_name();
            static std::basic_string<CharT> long_name();
        };
    }
se><phra
```

Rational Constant

Rational Constant Concept

Description

A Rational Constant is a holder class for a compile-time value of a rational type. Every Rational Constant is also a nullary Metafunction, returning itself. A rational constant object is implicitly convertible to the corresponding run-time value of the rational type.

Expression requirements

In the following table and subsequent specifications, r is a model of Rational Constant.

Expression	Type	Complexity
`r::tag`	`rational_c_tag`	Constant time
`r::value_type`	A rational type	Constant time
`r::num_type`	An integral type	Constant time
`r::den_type`	An integral type	Constant time
`r::num`	An Integral constant expression	Constant time
`r::den`	An Integral constant expression	Constant time
`r::type`	Rational Constant	Constant time
`r::value_type const c=r()`		Constant time

Expression semantics

Expression	Semantics
`r::tag`	r's tag type; r::tag::value is r's conversion rank.
`r::value_type`	A cv-unqualified type of `r()`
`r::num_type`	A cv-unqualified type of `r::num`
`r::den_type`	A cv-unqualified type of `r::den`
`r::num`	The numerator of the rational constant
`r::den`	The denominator of the rational constant
`r::type`	equal_to<n::type,n>::value == true.
`r::value_type const c=r()`	`r::value_type const c=r::value_type(r::num,r::den)`

Models

- `ratio<>`

Header `<boost/ratio/mpl/rational_constant.hpp>`

This header includes all the rational constant related header files

```
    #include <boost/ratio/mpl/rational_c_tag.hpp>
    #include <boost/ratio/mpl/numeric_cast.hpp>
    #include <boost/ratio/mpl/arithmetic.hpp>
    #include <boost/ratio/mpl/comparison.hpp>
</em
```

Header `<boost/ratio/mpl/rational_c_tag.hpp>`

```
    namespace boost {
    namespace mpl {

        struct rational_c_tag : int_<10> {};

    }
    }
/phrase
```

Header <boost/ratio/mpl/numeric_cast.hpp>

```
    namespace boost {
    namespace mpl {

        template<> struct numeric_cast< integral_c_tag,rational_c_tag >;

    }
    }
></code
```

mpl::numeric_cast<> Specialization

A Integral Constant is seen as a ratio with numerator the Integral Constant value and denominator 1.

```
        template<> struct numeric_cast< integral_c_tag,rational_c_tag >
        {
            template< typename N > struct apply
                : ratio< N::value, 1 >
            {
            };
        };
#####u#
```

Header <boost/ratio/mpl/arithmetic.hpp>

This header includes all the rational constant arithmetic MPL specializations.

```
    #include <boost/ratio/mpl/plus.hpp>
    #include <boost/ratio/mpl/minus.hpp>
    #include <boost/ratio/mpl/times.hpp>
    #include <boost/ratio/mpl/divides.hpp>
    #include <boost/ratio/mpl/negate.hpp>
    #include <boost/ratio/mpl/abs.hpp>
    #include <boost/ratio/mpl/sign.hpp>
    #include <boost/ratio/mpl/gcd.hpp>
    #include <boost/ratio/mpl/lcm.hpp>
rase><phr
```

Header <boost/ratio/mpl/plus.hpp>

```
    namespace boost {
    namespace mpl {
        template<>
        struct plus_impl< rational_c_tag,rational_c_tag >;
    }
    }
e=#spe
```

mpl::plus_impl<> Specialization

The specialization relays on the `ratio_add` template class.

```
        template<>
        struct plus_impl< rational_c_tag,rational_c_tag >
        {
            template< typename R1, typename R2 > struct apply
                : ratio_add<R1, R2>
            {
            };
        };
hrase ro
```

Header <boost/ratio/mpl/minus.hpp>

```
    namespace boost {
    namespace mpl {
        template<>
        struct minus_impl< rational_c_tag,rational_c_tag >;
    }
    }
pecial
```

mpl::minus_impl<> Specialization

The specialization relays on the `ratio_subtract` template class.

```
        template<>
        struct plus_impl< rational_c_tag,rational_c_tag >
        {
            template< typename R1, typename R2 > struct apply
                : ratio_subtract<R1, R2>
            {
            };
        };
<phrase ↵
```

Header <boost/ratio/mpl/times.hpp>

```
    namespace boost {
    namespace mpl {
        template<>
        struct times_impl< rational_c_tag,rational_c_tag >;
    }
    }
specia
```

mpl::times_impl<> Specialization

The specialization relays on the `ratio_multiply` template class.

```
        template<>
        struct times_impl< rational_c_tag,rational_c_tag >
        {
            template< typename R1, typename R2 > struct apply
                : ratio_multiply<R1, R2>
            {
            };
        };
ro#e=#id
```

Header <boost/ratio/mpl/divides.hpp>

```
    namespace boost {
    namespace mpl {
        template<>
        struct divides_impl< rational_c_tag,rational_c_tag >;
    }
    }
ecial#
```

mpl::divides_impl<> Specialization

The specialization relays on the `ratio_divide` template class.

```
        template<>
        struct divides_impl< rational_c_tag,rational_c_tag >
        {
            template< typename R1, typename R2 > struct apply
                : ratio_divide<R1, R2>
            {
            };
        };
phrase #
```

Header <boost/ratio/mpl/gcd.hpp>

```
    namespace boost {
    namespace mpl {
        template<>
        struct gcd_impl< rational_c_tag,rational_c_tag >;
    }
    }
le=#sp
```

mpl::gcd_impl<> Specialization

The specialization relays on the `ratio_gcd` template class.

```
        template<>
        struct gcd_impl< rational_c_tag,rational_c_tag >
        {
            template< typename R1, typename R2 > struct apply
                : ratio_gcd<R1, R2>
            {
            };
        };
};
r
```

Header <boost/ratio/mpl/lcm.hpp>

```
    namespace boost {
    namespace mpl {
        template<>
        struct lcm_impl< rational_c_tag,rational_c_tag >;
    }
    }
specia
```

mpl::lcm_impl<> Specialization

The specialization relays on the `ratio_lcm` template class.

```
        template<>
        struct lcm_impl< rational_c_tag,rational_c_tag >
        {
            template< typename R1, typename R2 > struct apply
                : ratio_lcm<R1, R2>
            {
            };
        };
};
p
```

Header <boost/ratio/mpl/negate.hpp>

```
    namespace boost {
    namespace mpl {
        template<>
        struct negate_impl< rational_c_tag >;
    }
    }
hrase ↵
```

mpl::negate_impl<> Specialization

The specialization relays on the `ratio_negate` template class.

```
        template<>
        struct negate_impl< rational_c_tag >
        {
            template< typename R > struct apply
                : ratio_negate<R>
            {
            };
        };
except  ↵
```

Header `<boost/ratio/mpl/abs.hpp>`

```
    namespace boost {
    namespace mpl {
        template<>
        struct abs_impl< rational_c_tag >;
    }
    }
#ident
```

`mpl::abs_impl<>` Specialization

The specialization relays on the `ratio_abs` template class.

```
        template<>
        struct abs_impl< rational_c_tag >
        {
            template< typename R > struct apply
                : ratio_abs<R>
            {
            };
        };
;
      ↵
```

Header `<boost/ratio/mpl/sign.hpp>`

```
    namespace boost {
    namespace mpl {
        template<>
        struct sign_impl< rational_c_tag >;
    }
    }
identi
```

`mpl::sign_impl<>` Specialization

The specialization relays on the `ratio_sign` template class.

```
        template<>
        struct sign_impl< rational_c_tag >
        {
            template< typename R > struct apply
                : ratio_sign<R>
            {
            };
        };
exce
```

Header <boost/ratio/mpl/comparison.hpp>

This header includes all the rational constant comparison MPL specializations.

```
    #include <boost/ratio/mpl/equal_to.hpp>
    #include <boost/ratio/mpl/not_equal_to.hpp>
    #include <boost/ratio/mpl/less.hpp>
    #include <boost/ratio/mpl/less_equal.hpp>
    #include <boost/ratio/mpl/greater.hpp>
    #include <boost/ratio/mpl/greater_equal.hpp>
se rol
```

Header <boost/ratio/mpl/equal_to.hpp>

```
    namespace boost {
    namespace mpl {
        template<>
        struct equal_to_impl< rational_c_tag,rational_c_tag >;
    }
    }
ign</p
```

mpl::equal_to_impl<> Specialization

The specialization relays on the `ratio_equal` template class.

```
        template<>
        struct equal_to_impl< rational_c_tag,rational_c_tag >
        {
            template< typename R1, typename R2 > struct apply
                : ratio_equal<R1, R2>
            {
            };
        };
p#rase #
```

Header `<boost/ratio/mpl/not_equal_to.hpp>`

```
    namespace boost {
    namespace mpl {
        template<>
        struct not_equal_to_impl< rational_c_tag,rational_c_tag >;
    }
    }
ial#>&
```

`mpl::not_equal_to_impl<>` Specialization

The specialization relays on the `ratio_not_equal` template class.

```
        template<>
        struct not_equal_to_impl< rational_c_tag,rational_c_tag >
        {
            template< typename R1, typename R2 > struct apply
                : ratio_not_equal<R1, R2>
            {
            };
        };
o/mpl/gr
```

Header `<boost/ratio/mpl/less.hpp>`

```
    namespace boost {
    namespace mpl {
        template<>
        struct less_impl< rational_c_tag,rational_c_tag >;
    }
    }
e role
```

`mpl::less_impl<>` Specialization

The specialization relays on the `ratio_less` template class.

```
        template<>
        struct less_impl< rational_c_tag,rational_c_tag >
        {
            template< typename R1, typename R2 > struct apply
                : ratio_less<R1, R2>
            {
            };
        };
};
p
```

Header `<boost/ratio/mpl/less_equal.hpp>`

```
    namespace boost {
    namespace mpl {
        template<>
        struct less_equal_impl< rational_c_tag,rational_c_tag >;
    }
    }
l#>&lt
```

`mpl::less_equal_impl<>` Specialization

The specialization relays on the `ratio_less_equal` template class.

```
        template<>
        struct less_equal_impl< rational_c_tag,rational_c_tag >
        {
            template< typename R1, typename R2 > struct apply
                : ratio_less_equal<R1, R2>
            {
            };
        };

o/mpl
```

Header `<boost/ratio/mpl/greater.hpp>`

```
    namespace boost {
    namespace mpl {
        template<>
        struct greater_impl< rational_c_tag,rational_c_tag >;
    }
    }
e=#spe
```

`mpl::greater_impl<>` Specialization

The specialization relays on the `ratio_greater` template class.

```
        template<>
        struct greater_impl< rational_c_tag,rational_c_tag >
        {
            template< typename R1, typename R2 > struct apply
                : ratio_greater<R1, R2>
            {
            };
        };

p#rase ↵
```

Header `<boost/ratio/mpl/greater_equal.hpp>`

```
    namespace boost {
    namespace mpl {
        template<>
        struct greater_equal_impl< rational_c_tag,rational_c_tag >;
    }
    }
l#>&lt
```

`mpl::greater_equal_impl<>` Specialization

The specialization relays on the `ratio_greater_equal` template class.

```
        template<>
        struct greater_equal_impl< rational_c_tag,rational_c_tag >
        {
            template< typename R1, typename R2 > struct apply
                : ratio_greater_equal<R1, R2>
            {
            };
        };
pl#gr#at
```

Appendices

Appendix A: History

Version 1.0.1, Jan 8, 2011

- Added MPL Rational Constant and the associated numeric metafunction specializations.

Version 1.0.0, Jan 2, 2011

- Moved ratio to trunk.
- Documentation revision.

Version 0.2.1, September 27, 2010

Fixes:

- Removal of LLVM adapted files due to incompatible License issue.

Version 0.2.0, September 22, 2010

Features:

- Added ratio_string traits.

Fixes:

- ratio_less overflow avoided following the algorithm from libc++.

Test:

- A more complete test has been included adapted from the test of from libc++/ratio.

Version 0.1.0, September 10, 2010

Features:

- Ratio has been extracted from Boost.Chrono.

Appendix B: Rationale

Why ratio needs CopyConstruction and Assignment from ratios having the same normalized form

Current **N3000** doesn't allows to copy-construct or assign ratio instances of ratio classes having the same normalized form.

This simple example

```
    ratio<1,3> r1;
    ratio<3,9> r2;
    r1 = r2; // (1)
vin
```

fails to compile in (1). Other example

```
    ratio<1,3> r1;
    ratio_subtract<ratio<2,3>,ratio<1,3> > r2=r1;  // (2)
##
```

The type of `ratio_subtract<ratio<2,3>,ratio<1,3> >` could be `ratio<3,9>` so the compilation could fail in (2). It could also be `ratio`<1,3> and the compilation succeeds.

Why ratio needs the nested normalizer typedef type

The current resolution of issue LWG 1281 acknowledges the need for a nested type typedef, so Boost.Ratio is tracking the likely final version of std::ratio.

Appendix C: Implementation Notes

How does Boost.Ratio try to avoid compile-time rational arithmetic overflow?

When the result is representable, but a simple application of arithmetic rules would result in overflow, e.g. `ratio_multiply<ratio<INTMAX_MAX,2>,ratio<2,INTMAX_MAX>>` can be reduced to `ratio<1,1>`, but the direct result of `ratio<INTMAX_MAX*2,INTMAX_MAX*2>` would result in overflow.

Boost.Ratio implements some simplifications in order to reduce the possibility of overflow. The general ideas are:

- The `num` and `denratio<>` fields are normalized.
- Use the gcd of some of the possible products that can overflow, and simplify before doing the product.
- Use some equivalences relations that avoid addition or subtraction that can overflow or underflow.

The following subsections cover each case in more detail.

ratio_add

In

```
    (n1/d1)+(n2/d2)=(n1*d2+n2*d1)/(d1*d2)
#
```

either n1*d2+n2*d1 or d1*d2 can overflow.

```
      ( (n1 * d2)  + (n2 * d1) )
      --------------------------
               (d1 * d2)
low
```

Dividing by gcd(d1,d2) on both num and den

```
      ( (n1 * (d2/gcd(d1,d2)))  + (n2 * (d1/gcd(d1,d2))) )
      ----------------------------------------------------
                     ((d1 * d2) / gcd(d1,d2))
phr
```

Multipliying and diving by gcd(n1,n2) in numerator

```
     ( ((gcd(n1,n2)*(n1/gcd(n1,n2))) * (d2/gcd(d1,d2)))  +
       ((gcd(n1,n2)*(n2/gcd(n1,n2))) * (d1/gcd(d1,d2)))
     )
     --------------------------------------------------
              ( (d1 * d2) / gcd(d1,d2) )
role
```

Factorizing gcd(n1,n2)

```
      ( gcd(n1,n2) *
        ( ((n1/gcd(n1,n2)) * (d2/gcd(d1,d2))) + ((n2/gcd(n1,n2)) * (d1/gcd(d1,d2))) )
      )
      --------------------------------------------------------------------------------
                                 ( (d1 * d2) / gcd(d1,d2) )
>rati
```

Regrouping

```
      ( gcd(n1,n2) *
        ( ((n1/gcd(n1,n2)) * (d2/gcd(d1,d2))) + ((n2/gcd(n1,n2)) * (d1/gcd(d1,d2))) )
      )
      --------------------------------------------------------------------------------
                               ( (d1 / gcd(d1,d2)) * d2 )
le=#s
```

Dividing by (d1 / gcd(d1,d2))

```
      ( ( gcd(n1,n2) / (d1 / gcd(d1,d2)) ) *
        ( ((n1/gcd(n1,n2)) * (d2/gcd(d1,d2))) + ((n2/gcd(n1,n2)) * (d1/gcd(d1,d2))) )
      )
      -------------------------------------------------------------------------------
                                              d2
ase><
```

Dividing by d2

```
    ( gcd(n1,n2) / (d1 / gcd(d1,d2)) ) *
    ( ((n1/gcd(n1,n2)) * (d2/gcd(d1,d2))) + ((n2/gcd(n1,n2)) * (d1/gcd(d1,d2))) / d2 )
e=
```

This expression correspond to the multiply of two ratios that have less risk of overflow as the initial numerators and denominators appear now in most of the cases divided by a gcd.

For ratio_subtract the reasoning is the same.

ratio_multiply

In

```
    (n1/d1)*(n2/d2)=((n1*n2)/(d1*d2))
#
```

either n1*n2 or d1*d2 can overflow.

Dividing by gcc(n1,d2) numerator and denominator

```
        (((n1/gcc(n1,d2))*n2)
        ---------------------
        (d1*(d2/gcc(n1,d2))))
#ap
```

Dividing by gcc(n2,d1)

```
       ((n1/gcc(n1,d2))*(n2/gcc(n2,d1)))
       ---------------------------------
       ((d1/gcc(n2,d1))*(d2/gcc(n1,d2)))
)
```

And now all the initial numerator and denominators have been reduced, avoiding the overflow.

For ratio_divide the reasoning is similar.

ratio_less

In order to evaluate

```
    (n1/d1)<(n2/d2)
(
```

without moving to floating-point numbers, two techniques are used:

- First compare the sign of the numerators.

If sign(n1) < sign(n2) the result is true.

If sign(n1) == sign(n2) the result depends on the following after making the numerators positive

- When the sign is equal the technique used is to work with integer division and modulo when the signs are equal.

Let call Qi the integer division of ni and di, and Mi the modulo of ni and di.

```
    ni = Qi * di + Mi and Mi < di
.
```

Form

```
    ((n1*d2)<(d1*n2))
#
```

we get

```
    (((Q1 * d1 + M1)*d2)<(d1*((Q2 * d2 + M2))))
#
```

Developing

```
    ((Q1 * d1 * d2)+ (M1*d2))<((d1 * Q2 * d2) + (d1*M2))
#
```

Dividing by d1*d2

```
    Q1 + (M1/d1) < Q2 + (M2/d2)
i
```

If Q1=Q2 the result depends on

```
    (M1/d1) < (M2/d2)
#
```

If M1=`0`=M2 the result is false

If M1=0 M2!=0 the result is true

If M1!`0 M2`=0 the result is false

If M1!=0 M2!=0 the result depends on

```
    (d2/M2) < (d1/M1)
#
```

If Q1!=Q2, the result of

```
    Q1 + (M1/d1) < Q2 + (M2/d2)
s
```

depends only on Q1 and Q2 as Qi are integers and (Mi/di) <1 because Mi<di.

if Q1>Q2, Q1==Q2+k, k>=1

```
    Q2+k + (M1/d1) < Q2 + (M2/d2)
    k + (M1/d1) < (M2/d2)
    k < (M2/d2) - (M1/d1)
flo
```

but the difference between two numbers between 0 and 1 can not be greater than 1, so the result is false.

if Q2>Q1, Q2==Q1+k, k>=1

```
    Q1 + (M1/d1) < Q1+k + (M2/d2)
    (M1/d1) < k + (M2/d2)
    (M1/d1) - (M2/d2) < k

fl
```

which is always true, so the result is true.

The following table recapitulates this analisys

ratio<n1,d1>	**ratio<n2,d2>**	**Q1**	**Q2**	**M1**	**M2**	**Result**
ratio<n1,d1>	ratio<n2,d2>	Q1	Q2	!=0	!=0	Q1 < Q2
ratio<n1,d1>	ratio<n2,d2>	Q	Q	0	0	false
ratio<n1,d1>	ratio<n2,d2>	Q	Q	0	!=0	true
ratio<n1,d1>	ratio<n2,d2>	Q	Q	!=0	0	false
ratio<n1,d1>	ratio<n2,d2>	Q	Q	!=0	!=0	ratio_less<ratio<d2,M2>, ratio<d1/M1>>

Appendix D: FAQ

Appendix E: Acknowledgements

The library code was derived from Howard Hinnant's `time2_demo` prototype. Many thanks to Howard for making his code available under the Boost license. The original code was modified by Beman Dawes to conform to Boost conventions.

`time2_demo` contained this comment:

Much thanks to Andrei Alexandrescu, Walter Brown, Peter Dimov, Jeff Garland, Terry Golubiewski, Daniel Krugler, Anthony Williams.

Howard Hinnant, who is the real author of the library, has provided valuable feedback and suggestions during the development of the library. In particular, The ratio_io.hpp source has been adapted from the experimental header `<ratio_io>` from Howard Hinnant.

The acceptance review of Boost.Ratio took place between October 2nd and 11th 2010. Many thanks to Anthony Williams, the review manager, and to all the reviewers: Bruno Santos, Joel Falcou, Robert Stewart, Roland Bock, Tom Tan and Paul A. Bristol.

Thanks to Andrew Chinoff and Paul A. Bristol for his help polishing the documentation.

Appendix F: Tests

In order to test you need to run

```
bjam libs/ratio/test
```

You can also run a specific suite of test by doing

```
    cd libs/chrono/test
    bjam ratio
hr
```

`ratio`

Name	kind	Description	Result	Ticket
typedefs.pass	run	check the num/den are correct for the predefined typedefs	Pass	#
ratio.pass	run	check the num/den are correctly simplified	Pass	#
ratio1.fail	compile-fails	The template argument D shall not be zero	Pass	#
ratio2.fail	compile-fails	the absolute values of the template arguments N and D shall be representable by type intmax_t	Pass	#
ratio3.fail	compile-fails	the absolute values of the template arguments N and D shall be representable by type intmax_t	Pass	#

`comparison`

Name	kind	Description	Result	Ticket
ratio_equal.pass	run	check ratio_equal metafunction class	Pass	#
ratio_not_equal.pass	run	check ratio_not_equal metafunction class	Pass	#
ratio_less.pass	run	check ratio_less metafunction class	Pass	#
ratio_less_equal.pass	run	check ratio_less_equal metafunction class	Pass	#
ratio_greater.pass	run	check ratio_greater metafunction class	Pass	#
ratio_greater_equal.pass	run	check ratio_greater_equal metafunction class	Pass	#

arithmetic

Name	kind	Description	Result	Ticket
ratio_add.pass	run	check ratio_add metafunction class	Pass	#
ratio_subtract.pass	run	check ratio_subtract metafunction class	Pass	#
ratio_multiply.pass	run	check ratio_multiply metafunction class	Pass	#
ratio_divide.pass	run	check ratio_divide metafunction class	Pass	#
ratio_add.fail	compile-fails	check ratio_add overflow metafunction class	Pass	#
ratio_subtract.fail	compile-fails	check ratio_subtract underflow metafunction class	Pass	#
ratio_multiply.fail	compile-fails	check ratio_multiply overflow metafunction class	Pass	#
ratio_divide.fail	compile-fails	check ratio_divide overflow metafunction class	Pass	#

Appendix G: Tickets

Ticket	Description	Resolution	State
1	result of metafunctions ratio_multiply and ratio_divide were not normalized ratios.	Use of the nested ratio typedef type on ratio arithmetic operations.	Closed
2	INTMAX_C is not always defined.	Replace INTMAX_C by BOOST_INTMAX_C until boost/cstdint.hpp ensures INTMAX_C is always defined.	Closed
3	MSVC reports a warning instead of an error when there is an integral constant overflow.	manage with MSVC reporting a warning instead of an error when there is an integral constant overflow.	Closed
4	ration_less overflow on cases where it can be avoided.	Change the algorithm as implemented in libc++.	Closed

Appendix H: Future Plans

For later releases

- Use template aliases on compiler providing it.
- Implement multiple arguments ratio arithmetic.