

rational

1.7

Generated by Doxygen 1.8.10

Wed Oct 21 2015 07:12:16

Contents

1	Module Index	1
1.1	Modules	1
2	Class Index	3
2.1	Class List	3
3	Module Documentation	5
3.1	CLN - Class Library for Numbers extensions	5
3.1.1	Detailed Description	5
3.1.2	Macro Definition Documentation	6
3.1.2.1	CLN_EPSILON	6
3.1.2.2	CLN_PRECISION	6
3.1.3	Typedef Documentation	6
3.1.3.1	cln_rational	6
3.2	Expression templates	7
3.2.1	Detailed Description	8
3.2.2	Function Documentation	9
3.2.2.1	abs(const Commons::Math::RationalExpression< T, A, GCD, CHKOP > &a)	9
3.2.2.2	abs(const Commons::Math::Rational< A, GCD, CHKOP > &a)	9
3.2.2.3	eval_rat_expr(const ExprT &expr, const typename RationalExpressionTraits< ExprT >::expr_type::result_type &val=typename RationalExpressionTraits< ExprT >::expr_type::result_type())	9
3.2.2.4	inv(const Commons::Math::RationalExpression< T, A, GCD, CHKOP > &a)	9
3.2.2.5	inv(const Commons::Math::Rational< A, GCD, CHKOP > &a)	10
3.2.2.6	mk_rat_lit(const Rational< T, GCD, CHKOP > &r)	10
3.2.2.7	mk_rat_proto_var(const RationalExpression< T, E, GCD, CHKOP > &)	10
3.2.2.8	mk_rat_proto_var(const Rational< T, GCD, CHKOP > &)	11
3.3	GNU Multiple Precisions extensions	12
3.3.1	Detailed Description	12
3.3.2	Macro Definition Documentation	12
3.3.2.1	GMP_EPSILON	12
3.3.3	Typedef Documentation	12
3.3.3.1	gmp_rational	12

3.4	Inflnt extensions	14
3.4.1	Detailed Description	14
3.4.2	Typedef Documentation	14
3.4.2.1	inflt_rational	14
3.5	Rational fraction class	15
3.5.1	Detailed Description	15
3.5.2	Function Documentation	16
3.5.2.1	modf(const Commons::Math::Rational< T, GCD, CHKOP > &__x, typename Commons::Math::Rational< T, GCD, CHKOP >::integer_type *__iptr)	16
3.6	Greatest common divisor algorithms	17
3.6.1	Detailed Description	17
4	Class Documentation	19
4.1	Commons::Math::ENABLE_OVERFLOW_CHECK< Op, T, IsSigned > Struct Template Reference	19
4.1.1	Detailed Description	19
4.2	Commons::Math::EPSILON< T > Struct Template Reference	19
4.2.1	Detailed Description	19
4.2.2	Member Function Documentation	20
4.2.2.1	value()	20
4.3	Commons::Math::GCD_cln< T, IsSigned, CHKOP, CONV > Struct Template Reference	20
4.3.1	Detailed Description	20
4.4	Commons::Math::GCD_euclid< T, IsSigned, CHKOP, CONV > Struct Template Reference	20
4.4.1	Detailed Description	21
4.5	Commons::Math::GCD_euclid_fast< T, IsSigned, CHKOP, CONV > Struct Template Reference	21
4.5.1	Detailed Description	21
4.6	Commons::Math::GCD_gmp< T, IsSigned, CHKOP, CONV > Struct Template Reference	21
4.6.1	Detailed Description	22
4.7	Commons::Math::GCD_stein< T, IsSigned, CHKOP, CONV > Struct Template Reference	22
4.7.1	Detailed Description	22
4.8	Commons::Math::NO_OPERATOR_CHECK< Op, T, IsSigned > Struct Template Reference	22
4.8.1	Detailed Description	22
4.9	Commons::Math::Rational< T, GCD, CHKOP > Class Template Reference	24
4.9.1	Detailed Description	28
4.9.2	Member Typedef Documentation	28
4.9.2.1	integer_type	28
4.9.2.2	mod_type	28
4.9.2.3	op_divides	28
4.9.2.4	op_minus	29
4.9.2.5	op_modulus	29
4.9.2.6	op_multiplies	29

4.9.2.7	op_negate	29
4.9.2.8	op_plus	29
4.9.3	Constructor & Destructor Documentation	29
4.9.3.1	Rational()	29
4.9.3.2	Rational(const Rational &other)	29
4.9.3.3	Rational(const Rational< integer_type, U1, V1 > &numer, const Rational< integer_type, U2, V2 > &denom)	30
4.9.3.4	Rational(const integer_type &numer, const integer_type &denom)	30
4.9.3.5	Rational(const integer_type &whole, const integer_type &numer, const integer_type &denom)	30
4.9.3.6	Rational(const NumberType &number)	31
4.9.3.7	Rational(const char *expr)	31
4.9.4	Member Function Documentation	32
4.9.4.1	abs() const	32
4.9.4.2	denominator() const NOEXCEPT	32
4.9.4.3	inverse() const	32
4.9.4.4	invert()	32
4.9.4.5	mod() const	33
4.9.4.6	numerator() const NOEXCEPT	33
4.9.4.7	operator NumberType() const	33
4.9.4.8	operator!() const	33
4.9.4.9	operator!=(const Rational &other) const	34
4.9.4.10	operator%(const Rational &other) const	34
4.9.4.11	operator%=(const Rational &other)	34
4.9.4.12	operator*(const Rational &other) const	35
4.9.4.13	operator*=(const Rational &other)	36
4.9.4.14	operator+(const Rational &other) const	36
4.9.4.15	operator+() const	36
4.9.4.16	operator++()	36
4.9.4.17	operator++(int)	37
4.9.4.18	operator+=(const Rational &other)	37
4.9.4.19	operator-(const Rational &other) const	37
4.9.4.20	operator-() const	37
4.9.4.21	operator--()	38
4.9.4.22	operator--(int)	38
4.9.4.23	operator-=(const Rational &other)	38
4.9.4.24	operator/(const Rational &other) const	38
4.9.4.25	operator/=(const Rational &other)	39
4.9.4.26	operator<(const Rational &other) const	39
4.9.4.27	operator<=(const Rational &other) const	39

4.9.4.28	operator=(const Rational &another)	39
4.9.4.29	operator=(const NumberType &number)	40
4.9.4.30	operator==(const Rational &other) const	40
4.9.4.31	operator>(const Rational &other) const	40
4.9.4.32	operator>=(const Rational &other) const	41
4.9.4.33	str(bool mixed=false) const	42
4.9.5	Friends And Related Function Documentation	42
4.9.5.1	operator!=(const NumberType &n, const Rational< T, GCD, CHKOP > &o)	42
4.9.5.2	operator!=(const Rational< T, GCD, CHKOP > &o, const NumberType &n)	42
4.9.5.3	operator%(const Rational< T, GCD, CHKOP > &o, const NumberType &n)	42
4.9.5.4	operator%(const NumberType &n, const Rational< T, GCD, CHKOP > &o)	43
4.9.5.5	operator%=(NumberType &n, const Rational< T, GCD, CHKOP > &o)	43
4.9.5.6	operator*(const Rational< T, GCD, CHKOP > &o, const NumberType &n)	43
4.9.5.7	operator*(const NumberType &n, const Rational< T, GCD, CHKOP > &o)	44
4.9.5.8	operator*=(NumberType &n, const Rational< T, GCD, CHKOP > &o)	44
4.9.5.9	operator+(const Rational< T, GCD, CHKOP > &o, const NumberType &n)	44
4.9.5.10	operator+(const NumberType &n, const Rational< T, GCD, CHKOP > &o)	44
4.9.5.11	operator+=(NumberType &n, const Rational< T, GCD, CHKOP > &o)	45
4.9.5.12	operator-(const Rational< T, GCD, CHKOP > &o, const NumberType &n)	45
4.9.5.13	operator-(const NumberType &n, const Rational< T, GCD, CHKOP > &o)	45
4.9.5.14	operator-=(NumberType &n, const Rational< T, GCD, CHKOP > &o)	46
4.9.5.15	operator/(const Rational< T, GCD, CHKOP > &o, const NumberType &n)	46
4.9.5.16	operator/(const NumberType &n, const Rational< T, GCD, CHKOP > &o)	46
4.9.5.17	operator/=(NumberType &n, const Rational< T, GCD, CHKOP > &o)	46
4.9.5.18	operator<(const NumberType &n, const Rational< T, GCD, CHKOP > &o)	47
4.9.5.19	operator<(const Rational< T, GCD, CHKOP > &o, const NumberType &n)	47
4.9.5.20	operator<<	47
4.9.5.21	operator<=(const NumberType &n, const Rational< T, GCD, CHKOP > &o)	48
4.9.5.22	operator<=(const Rational< T, GCD, CHKOP > &o, const NumberType &n)	48
4.9.5.23	operator==(const NumberType &n, const Rational< T, GCD, CHKOP > &o)	48
4.9.5.24	operator==(const Rational< T, GCD, CHKOP > &o, const NumberType &n)	49
4.9.5.25	operator>(const NumberType &n, const Rational< T, GCD, CHKOP > &o)	50
4.9.5.26	operator>(const Rational< T, GCD, CHKOP > &o, const NumberType &n)	50
4.9.5.27	operator>=(const NumberType &n, const Rational< T, GCD, CHKOP > &o)	50
4.9.5.28	operator>=(const Rational< T, GCD, CHKOP > &o, const NumberType &n)	51
4.9.5.29	operator>>	51
4.9.6	Member Data Documentation	51
4.9.6.1	one_	51
4.9.6.2	zero_	51
4.10	Commons::Math::RationalExpressionTraits< ExprT > Struct Template Reference	52

4.10.1 Detailed Description	52
4.10.2 Member Typedef Documentation	52
4.10.2.1 <code>expr_type</code>	52
4.10.2.2 <code>literal_type</code>	52
4.10.2.3 <code>variable_type</code>	53
4.11 <code>Commons::Math::TYPE_CONVERT< T ></code> Struct Template Reference	53
4.11.1 Detailed Description	53
4.11.2 Constructor & Destructor Documentation	53
4.11.2.1 <code>TYPE_CONVERT(const T &v)</code>	53
4.11.3 Member Function Documentation	53
4.11.3.1 <code>convert() const</code>	53
Index	55

Chapter 1

Module Index

1.1 Modules

Here is a list of all modules:

CLN - Class Library for Numbers extensions	5
Expression templates	7
GNU Multiple Precisions extensions	12
Inflnt extensions	14
Rational fraction class	15
Greatest common divisor algorithms	17

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

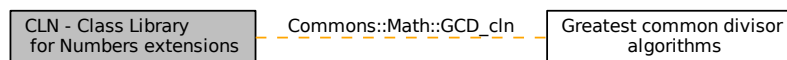
Commons::Math::ENABLE_OVERFLOW_CHECK< Op, T, IsSigned >	
Checked operator	19
Commons::Math::EPSILON< T >	
EPSILON for float approximation	19
Commons::Math::GCD_cln< T, IsSigned, CHKOP, CONV >	
CLN GCD algorithm	20
Commons::Math::GCD_euclid< T, IsSigned, CHKOP, CONV >	
Euclid GCD algorithm (safe) implementation	20
Commons::Math::GCD_euclid_fast< T, IsSigned, CHKOP, CONV >	
Euclid GCD algorithm (fast) implementation	21
Commons::Math::GCD_gmp< T, IsSigned, CHKOP, CONV >	
GMP GCD algorithm	21
Commons::Math::GCD_stein< T, IsSigned, CHKOP, CONV >	
Stein GCD algorithm implementation	22
Commons::Math::NO_OPERATOR_CHECK< Op, T, IsSigned >	
Unchecked operator	22
Commons::Math::Rational< T, GCD, CHKOP >	
Rational (fraction) template class	24
Commons::Math::RationalExpressionTraits< ExprT >	
Traits struct for expression templates	52
Commons::Math::TYPE_CONVERT< T >	
Type conversion policy class	53

Chapter 3

Module Documentation

3.1 CLN - Class Library for Numbers extensions

Collaboration diagram for CLN - Class Library for Numbers extensions:



Classes

- struct `Commons::Math::GCD_cln< T, IsSigned, CHKOP, CONV >`
CLN GCD algorithm.

Macros

- `#define CLN_PRECISION "30"`
- `#define CLN_EPSILON "1L-16_" CLN_PRECISION`

Typedefs

- typedef `Rational< cln::cl_I, Commons::Math::GCD_cln, Commons::Math::NO_OPERATOR_CHECK >`
`Commons::Math::cln_rational`
Rational class based on the CLN library.

3.1.1 Detailed Description

The header `cln_rational.h` contains specializations especially for the `CLN - Class Library for Numbers` as underlying storage type.

If you use the *CLN extensions*, you'll need to link your application with `-lcln`

3.1.2 Macro Definition Documentation

3.1.2.1 `#define CLN_EPSILON "1L-16_" CLN_PRECISION`

```
#include <cln_rational.h>
```

The `EPSILON` used for approximating a float

See also

[CLN_PRECISION](#)

[Commons::Math::EPSILON](#)

This define is passed as *is* to the constructor of `cln::cl_F`

3.1.2.2 `#define CLN_PRECISION "30"`

```
#include <cln_rational.h>
```

The default precision suffix

See also

[CLN_EPSILON](#)

[Commons::Math::EPSILON](#)

[Commons::Math::TYPE_CONVERT](#)

3.1.3 Typedef Documentation

3.1.3.1 `typedef Rational<cln::cl_I, Commons::Math::GCD_cln, Commons::Math::NO_OPERATOR_CHECK> Commons::Math::cln_rational`

```
#include <cln_rational.h>
```

[Rational](#) class based on the CLN library.

3.2 Expression templates

Classes

- struct `Commons::Math::RationalExpressionTraits< ExprT >`

Traits struct for expression templates.

Functions

- `template<class T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP>`
`RationalExpression< T, RationalConstant< T, GCD, CHKOP >, GCD, CHKOP >` `Commons::Math::mk_rat_lit` (const Rational< T, GCD, CHKOP > &r)
make a literal for use in expressions
- `template<class T , class E , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP>`
`RationalExpression< T, RationalVariable< T, GCD, CHKOP >, GCD, CHKOP >` `Commons::Math::mk_rat_proto_var` (const RationalExpression< T, E, GCD, CHKOP > &)
make a variable from a prototype
- `template<class T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP>`
`RationalExpression< T, RationalVariable< T, GCD, CHKOP >, GCD, CHKOP >` `Commons::Math::mk_rat_proto_var` (const Rational< T, GCD, CHKOP > &)
- `template<class ExprT >`
`RationalExpressionTraits< ExprT >::expr_type::result_type` `Commons::Math::eval_rat_expr` (const ExprT &expr, const typename RationalExpressionTraits< ExprT >::expr_type::result_type &val=typename RationalExpressionTraits< ExprT >::expr_type::result_type())
evaluates an expression
- `template<class T , class A , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP>`
`Commons::Math::RationalUnaryExpression< T, Commons::Math::RationalExpression< T, A, GCD, CHKOP >, Commons::Math::_unaryAbs< Commons::Math::RationalExpression< T, GCD, CHKOP > >, GCD, CHKOP >, GCD, CHKOP >` `abs` (const Commons::Math::RationalUnaryExpression< T, A, GCD, CHKOP > &a)
the absolute value of an expression
- `template<class T , class A , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP>`
`Commons::Math::RationalExpression< T, Commons::Math::RationalUnaryExpression< T, Commons::Math::RationalConstant< A, GCD, CHKOP >, Commons::Math::_unaryAbs< Commons::Math::RationalExpression< T, GCD, CHKOP > >, GCD, CHKOP >, GCD, CHKOP >` `abs` (const Commons::Math::RationalUnaryExpression< T, A, GCD, CHKOP > &a)
- `template<class T , class A , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP>`
`Commons::Math::RationalExpression< T, Commons::Math::RationalUnaryExpression< T, Commons::Math::RationalExpression< T, A, GCD, CHKOP >, Commons::Math::_unaryInv< Commons::Math::RationalExpression< T, GCD, CHKOP > >, GCD, CHKOP >, GCD, CHKOP >` `inv` (const Commons::Math::RationalUnaryExpression< T, A, GCD, CHKOP > &a)
inverts an expression
- `template<class T , class A , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP>`
`Commons::Math::RationalUnaryExpression< T, Commons::Math::RationalConstant< A, GCD, CHKOP >, Commons::Math::_unaryInv< Commons::Math::RationalExpression< T, GCD, CHKOP > >, GCD, CHKOP >, GCD, CHKOP >` `inv` (const Commons::Math::RationalUnaryExpression< T, A, GCD, CHKOP > &a)

3.2.1 Detailed Description

The *expression templates module* allows you to write your code in *domain-specific expressions* (DSE) as well as to lazy evaluate them.

Caveats:

- With *C++03* it is **not** possible to store expressions, but only to use them directly where expressions are expected as parameters.
With *C++11* you can use the keyword `auto` to store expressions
- Number literals are only possible at the right side of binary operators.
This literals get converted by using the *approximation* constructor `Commons::Math::Rational(const Number↵ Type &number)`

Example:

To approximate the integral $\int_a^b \frac{x}{1+x}$ by evaluating the expression $\frac{x}{1+x}$ for a specified number of equidistant points in the interval $[1, 5]$, using the [GNU Multiple Precisions extensions](#), you could write following `integrate()` template function:

```
template<class ExprT> inline static
Commons::Math::gmp_rational integrate ( const ExprT &e,
                                       const Commons::Math::gmp_rational &from,
                                       const Commons::Math::gmp_rational &to,
                                       std::size_t n ) {
    Commons::Math::gmp_rational sum;

    const static Commons::Math::gmp_rational two ( 2, 1 );
    const Commons::Math::gmp_rational &step ( ( to - from ) / n );

    for ( Commons::Math::gmp_rational i ( from + ( step / two ) ); i < to;
          i += step ) {
        sum += Commons::Math::eval_rat_expr ( e, i );
    }

    return step * sum;
}
```

next you can call the `integrate()` function like:

```
// declare the variable x from prototype gmp_rational
const RationalExpressionTraits<gmp_rational>::variable_type
&x ( mk_rat_proto_var ( gmp_rational() ) );

// approximate the integration for the interval [1, 5] using 10 equidistant points
const gmp_rational &r ( integrate ( x / ( x + 1 ), 1, 5, 10 ) );
```

This will yield the result $\frac{422563503196}{145568097675} \approx 2.9$ in `r`.

Note

You have to place the literal 1 as right operand, since *non-expressions* and *non-Rationals* cannot appear at the left side of an operator due to a restriction of the *C++* overload resolution. In this case it makes no difference due to the *commutative property* of the addition.

See also

[Commons::Math::RationalExpressionTraits](#)
[Commons::Math::mk_rat_lit\(\)](#)
[Commons::Math::mk_rat_proto_var\(\)](#)

3.2.2 Function Documentation

3.2.2.1 `template<class T, class A, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Commons::Math::RationalExpression<T, Commons::Math::RationalUnaryExpression<T, Commons::Math::RationalExpression<T, A, GCD, CHKOP>, Commons::Math::_unaryAbs<Commons::Math::Rational<T, GCD, CHKOP> >, GCD, CHKOP>, GCD, CHKOP> abs (const Commons::Math::RationalExpression< T, A, GCD, CHKOP > & a) [inline]`

```
#include <expr_rational.h>
```

the absolute value of an expression

See also

[Commons::Math::Rational::abs\(\)](#)

3.2.2.2 `template<class T, class A, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Commons::Math::RationalExpression<T, Commons::Math::RationalUnaryExpression<T, Commons::Math::RationalConstant<A, GCD, CHKOP>, Commons::Math::_unaryAbs<Commons::Math::Rational<T, GCD, CHKOP> >, GCD, CHKOP>, GCD, CHKOP> abs (const Commons::Math::Rational< A, GCD, CHKOP > & a) [inline]`

```
#include <expr_rational.h>
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

3.2.2.3 `template<class ExprT > RationalExpressionTraits<ExprT>::expr_type::result_type Commons::Math::eval_rat_expr (const ExprT & expr, const typename RationalExpressionTraits< ExprT >::expr_type::result_type & val = typename RationalExpressionTraits<ExprT>::expr_type::result_type()) [inline]`

```
#include <expr_rational.h>
```

evaluates an expression

Template Parameters

<i>ExprT</i>	the expression template type
--------------	------------------------------

Parameters

in	<i>expr</i>	the expression to evaluate
in	<i>val</i>	the value assigned to the variable

Returns

the result of the evaluated expression

3.2.2.4 `template<class T, class A, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Commons::Math::RationalExpression<T, Commons::Math::RationalUnaryExpression<T, Commons::Math::RationalExpression<T, A, GCD, CHKOP>, Commons::Math::_unaryInv<Commons::Math::Rational<T, GCD, CHKOP> >, GCD, CHKOP>, GCD, CHKOP> inv (const Commons::Math::RationalExpression< T, A, GCD, CHKOP > & a) [inline]`

```
#include <expr_rational.h>
```

inverts an expression

See also

[Commons::Math::Rational::inverse\(\)](#)

3.2.2.5 `template<class T, class A, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Commons::Math::RationalExpression<T, Commons::Math::RationalUnaryExpression<T, Commons::Math::RationalConstant<A, GCD, CHKOP>, Commons::Math::_unaryInv<Commons::Math::Rational<T, GCD, CHKOP> >, GCD, CHKOP>, GCD, CHKOP> inv (const Commons::Math::Rational< A, GCD, CHKOP > & a) [inline]`

```
#include <expr_rational.h>
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

3.2.2.6 `template<class T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> RationalExpression<T, RationalConstant<T, GCD, CHKOP>, GCD, CHKOP> Commons::Math::mk_rat_lit (const Rational< T, GCD, CHKOP > & r) [inline]`

```
#include <expr_rational.h>
```

make a literal for use in expressions

Example:

to create a literal `1` :

```
const Commons::Math::RationalExpressionTraits
<Commons::Math::gmp_rational>::expr_type
&l ( Commons::Math::mk_rat_lit (
Commons::Math::gmp_rational ( 1, 1 ) ) );
```

See also

[Commons::Math::RationalExpressionTraits](#)

Parameters

<i>r</i>	the Rational to create a literal for
----------	--

Returns

the literal for use in expressions

3.2.2.7 `template<class T, class E, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> RationalExpression<T, RationalVariable<T, GCD, CHKOP>, GCD, CHKOP> Commons::Math::mk_rat_proto_var (const RationalExpression< T, E, GCD, CHKOP > &) [inline]`

```
#include <expr_rational.h>
```

make a variable from a prototype

The prototype is needed for the compiler to deduce the right type

Example:

to create a variable `x` using the prototype `l` (a literal created by [mk_rat_lit\(\)](#) or by another [Rational](#)):

```
const Commons::Math::RationalExpressionTraits
<Commons::Math::gmp_rational>::variable_type
&x ( Commons::Math::mk_rat_proto_var ( l ) );
```

See also

[Commons::Math::RationalExpressionTraits](#)
[Commons::Math::mk_rat_lit\(\)](#)

Returns

a variable from a prototype

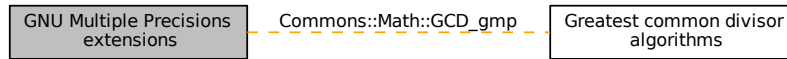
```
3.2.2.8 template<class T, template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD, template< class, typename, bool > class CHKOP> RationalExpression<T, RationalVariable<T,
GCD, CHKOP>, GCD, CHKOP> Commons::Math::mk_rat_proto_var ( const Rational< T, GCD, CHKOP > & )
[inline]
```

```
#include <expr_rational.h>
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

3.3 GNU Multiple Precisions extensions

Collaboration diagram for GNU Multiple Precisions extensions:



Classes

- struct [Commons::Math::GCD_gmp](#)< T, IsSigned, CHKOP, CONV >
GMP GCD algorithm.

Macros

- #define [GMP_EPSILON](#) "1e-21", 30, 10

Typedefs

- typedef Rational< mpz_class, GCD_gmp, NO_OPERATOR_CHECK > [Commons::Math::gmp_rational](#)
Rational class based on the GMP library.

3.3.1 Detailed Description

The header `gmp_rational.h` contains specializations especially for [the GNU Multiple Precision Arithmetic Library](#) as underlying storage type.

If you use the *GMP extensions*, you'll need to link your application with `-lgmpxx -lgmp`

3.3.2 Macro Definition Documentation

3.3.2.1 #define GMP_EPSILON "1e-21", 30, 10

```
#include <gmp_rational.h>
```

The `EPSILON` used for approximating a float

See also

[Commons::Math::EPSILON](#)

This define is passed as *is* to the constructor of `mpz_class`

See [C++ Interface Integers](#) for details

3.3.3 Typedef Documentation

3.3.3.1 typedef Rational<mpz_class, GCD_gmp, NO_OPERATOR_CHECK> Commons::Math::gmp_rational

```
#include <gmp_rational.h>
```

[Rational](#) class based on the GMP library.

3.4 InfInt extensions

Typedefs

- typedef Rational< InfInt, GCD_euclid > [Commons::Math::infint_rational](#)
Rational class based on InfInt.

3.4.1 Detailed Description

The header `infint_rational.h` contains specializations especially for `InfInt` as underlying as underlying storage type.

3.4.2 Typedef Documentation

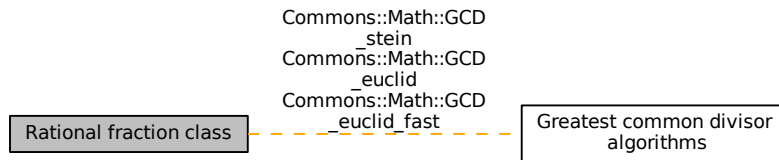
3.4.2.1 typedef Rational<InfInt, GCD_euclid> Commons::Math::infint_rational

```
#include <infint_rational.h>
```

[Rational](#) class based on InfInt.

3.5 Rational fraction class

Collaboration diagram for Rational fraction class:



Classes

- struct `Commons::Math::TYPE_CONVERT< T >`
Type conversion policy class.
- struct `Commons::Math::EPSILON< T >`
EPSILON for float approximation
- struct `Commons::Math::NO_OPERATOR_CHECK< Op, T, IsSigned >`
unchecked operator
- struct `Commons::Math::ENABLE_OVERFLOW_CHECK< Op, T, IsSigned >`
checked operator
- struct `Commons::Math::GCD_stein< T, IsSigned, CHKOP, CONV >`
Stein GCD algorithm implementation.
- struct `Commons::Math::GCD_euclid< T, IsSigned, CHKOP, CONV >`
Euclid GCD algorithm (safe) implementation.
- struct `Commons::Math::GCD_euclid_fast< T, IsSigned, CHKOP, CONV >`
Euclid GCD algorithm (fast) implementation.
- class `Commons::Math::Rational< T, GCD, CHKOP >`
Rational (fraction) template class

Functions

- `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP >`
`Commons::Math::Rational< T, GCD, CHKOP > std::modf (const Commons::Math::Rational< T, GCD, CHKOP > &__x, typename Commons::Math::Rational< T, GCD, CHKOP >::integer_type *__iptr)`
Overload of std::modf for Rational types.

3.5.1 Detailed Description

Include `rational.h` to be able to do fraction calculations. By simply including `rational.h` and specifying the storage type (any integer variant) you can create and use a fractional data type. For example, `Commons::Math::Rational<long> foo(3, 4)` would create a fraction named `foo` with a value of $\frac{3}{4}$, and store the fraction using the `long` data type.

Example:

To approximate the **reciprocal** of the *golden ratio* ($\varphi = \phi^{-1}$)

$$\varphi = \frac{\sqrt{5}-1}{2}$$

by iteratively calculating the continued fraction

$$\varphi = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \dots}}}}$$

you could write:

```
Rational<uint64_t> phi ( lu, lu );
for ( std::size_t i = 0u; i < 9lu; ++i ) ( ++phi ).invert();
```

which will result in $\varphi \approx \frac{7540113804746346429}{12200160415121876738} = 0.61803398874989484820458683436563811772$

Note

Use `Commons::Math::Rational::invert()` or just add 1 to get ϕ
 As *side effect* you'll get the *i*-th *Fibonacci number* in the numerator of each iteration

3.5.2 Function Documentation

3.5.2.1 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Commons::Math::Rational<T, GCD, CHKOP> std::modf (const Commons::Math::Rational< T, GCD, CHKOP > & __x, typename Commons::Math::Rational< T, GCD, CHKOP >::integer_type * __iptr) [inline]`

```
#include <rational.h>
```

Overload of `std::modf` for `Rational` types.

See also

`Rational::mod()`

Warning

`__iptr` **must** point to a valid address and **cannot** be `NULL`

Template Parameters

<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>NumberType</i>	the number type
<i>CHKOP</i>	checked operator

See also

`ENABLE_OVERFLOW_CHECK`

Parameters

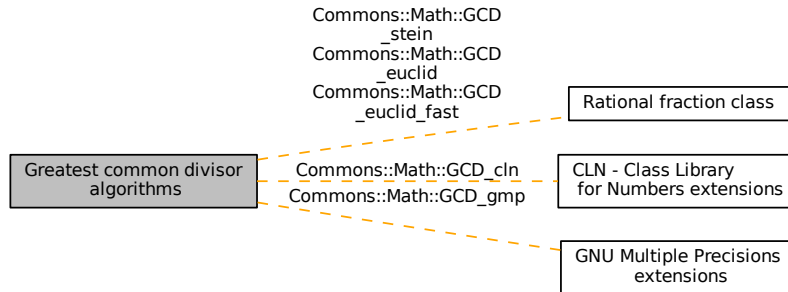
in	<code>__x</code>	the <code>Rational</code>
out	<code>__iptr</code>	address of the <code>integer_type</code> to store the integral part

Returns

the `Rational` part of `__x`

3.6 Greatest common divisor algorithms

Collaboration diagram for Greatest common divisor algorithms:



Classes

- struct [Commons::Math::GCD_cln](#)< T, IsSigned, CHKOP, CONV >
CLN GCD algorithm.
- struct [Commons::Math::GCD_gmp](#)< T, IsSigned, CHKOP, CONV >
GMP GCD algorithm.
- struct [Commons::Math::GCD_stein](#)< T, IsSigned, CHKOP, CONV >
Stein GCD algorithm implementation.
- struct [Commons::Math::GCD_euclid](#)< T, IsSigned, CHKOP, CONV >
Euclid GCD algorithm (safe) implementation.
- struct [Commons::Math::GCD_euclid_fast](#)< T, IsSigned, CHKOP, CONV >
Euclid GCD algorithm (fast) implementation.

3.6.1 Detailed Description

The *greatest common divisor algorithms* are used to reduce a [Commons::Math::Rational](#) so that $numerator \perp denominator$, i.e. $gcd(numerator, denominator) = 1$

Example:

A custom GCD algorithm could be implemented as following:

```
template<typename MyType, bool IsSigned, template<class, typename = MyType,
        bool = IsSigned> class CHKOP, template<typename = MyType> class CONV>
struct GCD_myType {
    inline MyType operator() ( const MyType &a, const MyType &b ) const {
        return myType_GCD_impl(a, b);
    }
};
```

and using it at second template parameter:

```
typedef Commons::Math::Rational<MyType, GCD_myType>
myType_rational;
```


Chapter 4

Class Documentation

4.1 Commons::Math::ENABLE_OVERFLOW_CHECK< Op, T, IsSigned > Struct Template Reference

checked operator

```
#include <rational.h>
```

4.1.1 Detailed Description

```
template<class Op, typename T, bool IsSigned>struct Commons::Math::ENABLE_OVERFLOW_CHECK< Op, T, IsSigned >
```

checked operator

Checks the operands on signed overflows, resp. unsigned wraps and throws a `std::domain_error` if the check fails, else delegates to the operator `Op`

Template Parameters

<i>Op</i>	operator functor
<i>T</i>	storage type
<i>IsSigned</i>	specialization for <i>signed</i> or <i>unsigned</i> types

4.2 Commons::Math::EPSILON< T > Struct Template Reference

EPSILON for float approximation

```
#include <rational.h>
```

Static Public Member Functions

- static const T `value` ()
the value of EPSILON

4.2.1 Detailed Description

```
template<typename T>struct Commons::Math::EPSILON< T >
```

EPSILON for float approximation

Specialize this class if you need another `EPSILON` (error tolerance)

By default this is

```
std::numeric_limits<T>::epsilon()
```

Template Parameters

<i>T</i>	storage type of Rational
----------	--

4.2.2 Member Function Documentation

4.2.2.1 `template<typename T > static const T Commons::Math::EPSILON< T >::value () [inline], [static]`

the value of `EPSILON`

4.3 Commons::Math::GCD_cln< T, IsSigned, CHKOP, CONV > Struct Template Reference

CLN GCD algorithm.

```
#include <cln_rational.h>
```

4.3.1 Detailed Description

```
template<typename T, bool IsSigned, template< class, typename=T, bool=IsSigned > class CHKOP, template< typename=T > class CONV> struct Commons::Math::GCD_cln< T, IsSigned, CHKOP, CONV >
```

CLN GCD algorithm.

The gcd algorithm implemented in the CLN library

Note

this is currently the **only** supported GCD for use with [CLN - Class Library for Numbers extensions](#)

Template Parameters

<i>T</i>	storage type
<i>IsSigned</i>	specialization for <i>signed</i> or <i>unsigned</i> types
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.4 Commons::Math::GCD_euclid< T, IsSigned, CHKOP, CONV > Struct Template Reference

Euclid GCD algorithm (safe) implementation.

```
#include <rational.h>
```

4.4.1 Detailed Description

```
template<typename T, bool IsSigned, template< class, typename=T, bool=IsSigned > class CHKOP, template< typename >
class CONV = TYPE_CONVERT>struct Commons::Math::GCD_euclid< T, IsSigned, CHKOP, CONV >
```

Euclid GCD algorithm (safe) implementation.

This implementation supports overflow/wrap checking

Template Parameters

<i>T</i>	storage type
<i>IsSigned</i>	specialization for <i>signed</i> or <i>unsigned</i> types
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.5 Commons::Math::GCD_euclid_fast< T, IsSigned, CHKOP, CONV > Struct Template Reference

Euclid GCD algorithm (fast) implementation.

```
#include <rational.h>
```

4.5.1 Detailed Description

```
template<typename T, bool IsSigned, template< class, typename=T, bool=IsSigned > class CHKOP, template< typename >
class CONV = TYPE_CONVERT>struct Commons::Math::GCD_euclid_fast< T, IsSigned, CHKOP, CONV >
```

Euclid GCD algorithm (fast) implementation.

See also

[GCD_euclid](#) if your number class doesn't support all needed operators

Template Parameters

<i>T</i>	storage type
<i>IsSigned</i>	specialization for <i>signed</i> or <i>unsigned</i> types
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.6 Commons::Math::GCD_gmp< T, IsSigned, CHKOP, CONV > Struct Template Reference

GMP GCD algorithm.

```
#include <gmp_rational.h>
```

4.6.1 Detailed Description

```
template<typename T, bool IsSigned, template< class, typename=T, bool=IsSigned > class CHKOP, template< typename=T >
class CONV>struct Commons::Math::GCD_gmp< T, IsSigned, CHKOP, CONV >
```

GMP GCD algorithm.

The gcd algorithm implemented in the GMP library

Template Parameters

<i>T</i>	storage type
<i>IsSigned</i>	specialization for <i>signed</i> or <i>unsigned</i> types
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.7 Commons::Math::GCD_stein< T, IsSigned, CHKOP, CONV > Struct Template Reference

Stein GCD algorithm implementation.

```
#include <rational.h>
```

4.7.1 Detailed Description

```
template<typename T, bool IsSigned, template< class, typename=T, bool=IsSigned > class CHKOP, template< typename >
class CONV = TYPE_CONVERT>struct Commons::Math::GCD_stein< T, IsSigned, CHKOP, CONV >
```

Stein GCD algorithm implementation.

Template Parameters

<i>T</i>	storage type
<i>IsSigned</i>	specialization for <i>signed</i> or <i>unsigned</i> types
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.8 Commons::Math::NO_OPERATOR_CHECK< Op, T, IsSigned > Struct Template Reference

unchecked operator

```
#include <rational.h>
```

4.8.1 Detailed Description

```
template<class Op, typename T, bool IsSigned>struct Commons::Math::NO_OPERATOR_CHECK< Op, T, IsSigned >
```

unchecked operator

Delegates the operator Op without any overflow/wrap check

Template Parameters

<i>Op</i>	operator functor
<i>T</i>	storage type
<i>IsSigned</i>	specialization for <i>signed</i> or <i>unsigned</i> types

4.9 Commons::Math::Rational< T, GCD, CHKOP > Class Template Reference

Rational (fraction) template class

```
#include <rational.h>
```

Public Types

- typedef T [integer_type](#)
storage type
- typedef [_mod](#)< [integer_type](#), GCD, CHKOP, std::numeric_limits< [integer_type](#) >::is_signed >::pair_type
[mod_type](#)
type of the return value of [mod\(\)](#)
- typedef CHKOP< std::plus< T > > [op_plus](#)
addition ($a + b$) operator
- typedef CHKOP< std::minus< T > > [op_minus](#)
subtraction ($a - b$) operator
- typedef CHKOP< std::negate< T > > [op_negate](#)
subtraction ($a - b$) operator
- typedef CHKOP< std::multiplies< T > > [op_multiplies](#)
*multiplication ($a * b$) operator*
- typedef CHKOP< std::divides< T > > [op_divides](#)
division (a / b) operator
- typedef CHKOP< std::modulus< T > > [op_modulus](#)
modulus ($a \% b$) operator

Public Member Functions

- [Rational](#) ()
creates a default (null) Rational
- [Rational](#) (const [Rational](#) &other)
copy constructor
- template<template< typename, bool, template< class, typename, bool > class, template< typename > class > class U1, template< class, typename, bool > class V1, template< typename, bool, template< class, typename, bool > class, template< typename > class > class U2, template< class, typename, bool > class V2>
[Rational](#) (const [Rational](#)< [integer_type](#), U1, V1 > &numer, const [Rational](#)< [integer_type](#), U2, V2 > &denom)
creates a Rational
- [Rational](#) (const [integer_type](#) &numer, const [integer_type](#) &denom)
creates a Rational
- [Rational](#) (const [integer_type](#) &whole, const [integer_type](#) &numer, const [integer_type](#) &denom)
creates a improper (mixed) Rational
- template<typename NumberType >
[Rational](#) (const NumberType &number)
creates an approximated Rational
- [Rational](#) (const char *expr)

- creates a Rational approximated by an expression*
- **Rational & operator=** (const **Rational** &another)
 - assign another Rational*
- template<typename NumberType >
 Rational & operator= (const NumberType &number)
 - assigns from a NumberType*
- template<typename NumberType >
operator NumberType () const
 - convert to NumberType*
- **integer_type numerator** () const NOEXCEPT
 - gets the numerator*
- **integer_type denominator** () const NOEXCEPT
 - gets the denominator*
- **mod_type mod** () const
 - extract the integral and fractional part*
- **Rational abs** () const
 - gets the absolute Rational*
- **Rational & invert** ()
 - inverts the Rational*
- **Rational inverse** () const
 - gets a copy of the inverted Rational*
- **Rational & operator+=** (const **Rational** &other)
 - add and assign a Rational*
- **Rational operator+** (const **Rational** &other) const
 - add a Rational*
- **Rational operator+** () const
 - get a copy of the Rational*
- **Rational & operator++** ()
 - pre-increment a Rational*
- **Rational operator++** (int)
 - post-increment a Rational*
- **Rational & operator-=** (const **Rational** &other)
 - subtract and assign a Rational*
- **Rational operator-** (const **Rational** &other) const
 - subtract a Rational*
- **Rational operator-** () const
 - get a negated copy of the Rational*
- **Rational & operator--** ()
 - pre-decrement a Rational*
- **Rational operator--** (int)
 - post-decrement a Rational*
- **Rational & operator*=** (const **Rational** &other)
 - multiply and assign a Rational*
- **Rational operator*** (const **Rational** &other) const
 - multiply a Rational*
- **Rational & operator/=** (const **Rational** &other)
 - divide and assign a Rational*
- **Rational operator/** (const **Rational** &other) const
 - divide a Rational*
- **Rational & operator%=** (const **Rational** &other)
 - modulo and assign a Rational*

- **Rational operator%** (const **Rational** &other) const
modulo a Rational
- **bool operator==** (const **Rational** &other) const
test on equality
- **bool operator!=** (const **Rational** &other) const
test on inequality
- **bool operator<** (const **Rational** &other) const
test if less than
- **bool operator<=** (const **Rational** &other) const
test if less or equal than
- **bool operator>** (const **Rational** &other) const
test if greater than
- **bool operator>=** (const **Rational** &other) const
test if greater or equal than
- **bool operator!** () const
test if it is the neutral element to addition and subtraction
- **std::string str** (bool mixed=false) const
generates the string representation of Rational

Static Public Attributes

- static const **integer_type zero_** = **integer_type**()
*represents zero in the given **Rational::integer_type***
- static const **integer_type one_** = **integer_type** (1)
*represents one in the given **Rational::integer_type***

Friends

- **std::ostream & operator<<** (std::ostream &o, const **Rational** &r)
output stream operator
- **std::istream & operator>>** (std::istream &i, **Rational** &r)
*reads in an expression from a **std::istream** and assign it to r*

Related Functions

(Note that these are not member functions.)

- **template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP>**
NumberType & operator+= (NumberType &n, const **Rational**< T, GCD, CHKOP > &o)
- **template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType >**
Rational< T, GCD, CHKOP > **operator+** (const **Rational**< T, GCD, CHKOP > &o, const NumberType &n)
- **template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP>**
Rational< T, GCD, CHKOP > **operator+** (const NumberType &n, const **Rational**< T, GCD, CHKOP > &o)
- **template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP>**
NumberType & operator-= (NumberType &n, const **Rational**< T, GCD, CHKOP > &o)
- **template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType >**
Rational< T, GCD, CHKOP > **operator-** (const **Rational**< T, GCD, CHKOP > &o, const NumberType &n)

- `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType > bool operator> (const Rational< T, GCD, CHKOP > &o, const NumberType &n)`
- `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> bool operator>= (const NumberType &n, const Rational< T, GCD, CHKOP > &o)`
- `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType > bool operator>= (const Rational< T, GCD, CHKOP > &o, const NumberType &n)`

4.9.1 Detailed Description

```
template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class >
class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_
_OPERATOR_CHECK>class Commons::Math::Rational< T, GCD, CHKOP >
```

Rational (fraction) template class

Note

All Rational objects are reduced (see [Greatest common divisor algorithms](#))

Template Parameters

<i>T</i>	storage type
<i>GCD</i>	GCD algorithm
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.2 Member Typedef Documentation

4.9.2.1 `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> typedef T Commons::Math::Rational< T, GCD, CHKOP >::integer_type`

storage type

4.9.2.2 `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> typedef _mod<integer_type, GCD, CHKOP, std::numeric_limits<integer_type>::is_signed>::pair_type Commons::Math::Rational< T, GCD, CHKOP >::mod_type`

type of the return value of `mod()`

This type is based on `std::pair`, where `first` is the integral value and `second` the fractional part

4.9.2.3 `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> typedef CHKOP<std::divides<T> > Commons::Math::Rational< T, GCD, CHKOP >::op_divides`

division (`a / b`) operator

4.9.2.4 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> typedef CHKOP<std::minus<T> > Commons::Math::Rational< T, GCD, CHKOP >::op_minus`

subtraction (a - b) operator

4.9.2.5 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> typedef CHKOP<std::modulus<T> > Commons::Math::Rational< T, GCD, CHKOP >::op_modulus`

modulus (a % b) operator

4.9.2.6 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> typedef CHKOP<std::multiplies<T> > Commons::Math::Rational< T, GCD, CHKOP >::op_multiplies`

multiplication (a * b) operator

4.9.2.7 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> typedef CHKOP<std::negate<T> > Commons::Math::Rational< T, GCD, CHKOP >::op_negate`

subtraction (a - b) operator

4.9.2.8 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> typedef CHKOP<std::plus<T> > Commons::Math::Rational< T, GCD, CHKOP >::op_plus`

addition (a + b) operator

4.9.3 Constructor & Destructor Documentation

4.9.3.1 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> Commons::Math::Rational< T, GCD, CHKOP >::Rational () [inline]`

creates a default (null) Rational

Creates a fraction $\frac{0}{1}$

4.9.3.2 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Commons::Math::Rational< T, GCD, CHKOP >::Rational (const Rational< T, GCD, CHKOP > & other)`

copy constructor

Parameters

<i>in</i>	<i>other</i>	the Rational to copy
-----------	--------------	----------------------

4.9.3.3 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> template<template< typename, bool, template< class, typename, bool > class, template< typename > class > class U1, template< class, typename, bool > class V1, template< typename, bool, template< class, typename, bool > class, template< typename > class > class U2, template< class, typename, bool > class V2> Commons::Math::Rational< T, GCD, CHKOP >::Rational (const Rational< integer_type, U1, V1 > & numer, const Rational< integer_type, U2, V2 > & denom) [inline]`

creates a Rational

Creates a copy of `numer` and divides it by `denom`

Note

to achieve a continued fraction like $\frac{1}{\frac{1}{2}}$ you must explicitly cast the *numer*, i.e.

```
const Rational<rational_type> x ( Rational<rational_type> ( 1 ),
    Rational<rational_type> ( 1,2 ));
```

Template Parameters

<i>U1</i>	GCD algorithm of the numerator
<i>V1</i>	operator checker of the numerator
<i>U2</i>	GCD algorithm of the denominator
<i>V2</i>	operator checker of the denominator

Parameters

<i>in</i>	<i>numer</i>	the numerator
<i>in</i>	<i>denom</i>	the denominator

4.9.3.4 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Commons::Math::Rational< T, GCD, CHKOP >::Rational (const integer_type & numer, const integer_type & denom)`

creates a Rational

Creates a fraction $\frac{numer}{denom}$

Parameters

<i>in</i>	<i>numer</i>	the numerator
<i>in</i>	<i>denom</i>	the denominator

4.9.3.5 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> Commons::Math::Rational< T, GCD, CHKOP >::Rational (const integer_type & whole, const integer_type & numer, const integer_type & denom) [inline]`

creates a improper (mixed) Rational

Creates a fraction $whole + \frac{numer}{denom}$

Parameters

<code>in</code>	<i>whole</i>	whole number part
<code>in</code>	<i>numer</i>	the numerator
<code>in</code>	<i>denom</i>	the denominator

4.9.3.6 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> template<typename NumberType> Commons::Math::Rational< T, GCD, CHKOP >::Rational (const NumberType & number)`

creates an approximated Rational

Example:

`Commons::Math::Rational<uint64_t>(std::sqrt(2.0l))` to get $\sqrt{2} \approx \frac{6333631924}{4478554083}$

See also

[Commons::Math::EPSILON](#) to control the quality of the approximation

Template Parameters

<i>NumberType</i>	type of the number to approximate
-------------------	-----------------------------------

Parameters

<code>in</code>	<i>number</i>	the number to create an approximated Rational of
-----------------	---------------	--

4.9.3.7 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Commons::Math::Rational< T, GCD, CHKOP >::Rational (const char * expr)`

creates a Rational approximated by an expression

If `expr` is not `null` and not empty then it is parsed and evaluated as `long double` expression and approximated to a fraction.

The *expression* can be a simple infix arithmetic expression containing

- addition (+); also *unary*
- subtraction (-); also *unary*
- multiplication (*)
- division (/)
- parentheses

Numbers can be integers or floats in non-scientific notation. Allowed are spaces and tabs around numbers, parentheses and operators.

The expression gets evaluated into an `long double` value and is then approximated to a fraction.

In case of errors an `std::runtime_exception` is thrown if exceptions are enabled, else the result is undefined.

Example:

```
Rational<long> x ( "(11/2) * +(4.25+3.75)" );
```

produces the fraction $x = \frac{44}{1}$

See also

[Rational\(const NumberType &number\)](#)

Parameters

<code>in</code>	<code>expr</code>	the expression to evaluate and approximate
-----------------	-------------------	--

4.9.4 Member Function Documentation

4.9.4.1 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> Rational Commons::Math::Rational< T, GCD, CHKOP >::abs ()`
`const [inline]`

gets the absolute Rational

Returns for *signed* types (`this->numerator() < T() ? -(*this) : (*this)`) and for *unsigned* types (`*this`)

Returns

a copy of the absolute Rational

4.9.4.2 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> integer_type Commons::Math::Rational< T, GCD, CHKOP >::denominator ()const [inline]`

gets the denominator

Returns

the denominator

4.9.4.3 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Rational< T, GCD, CHKOP > Commons::Math::Rational< T, GCD, CHKOP >::inverse () const`

gets a copy of the inverted Rational

See also

[invert\(\)](#)

Returns

a copy of the inverted Rational

4.9.4.4 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Rational< T, GCD, CHKOP > & Commons::Math::Rational< T, GCD, CHKOP >::invert ()`

inverts the Rational

Example:

$\frac{-5}{12}$ will become to $\frac{-12}{5}$

Returns

the inverted Rational

4.9.4.5 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Rational< T, GCD, CHKOP >::mod_type Commons::Math::Rational< T, GCD, CHKOP >::mod () const`

extract the integral and fractional part

See also

[mod_type](#)

Returns

a `mod_type` containing the integral and fractional part

4.9.4.6 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> integer_type Commons::Math::Rational< T, GCD, CHKOP >::numerator () const [inline]`

gets the numerator

Returns

the numerator

4.9.4.7 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> template<typename NumberType > Commons::Math::Rational< T, GCD, CHKOP >::operator NumberType () const [inline]`

convert to `NumberType`

Template Parameters

<i>NumberType</i>	type of the number to approximate
-------------------	-----------------------------------

Returns

the number value of the Rational

4.9.4.8 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> bool Commons::Math::Rational< T, GCD, CHKOP >::operator! () const [inline]`

test if it is the neutral element to addition and subtraction

Tests if the `numerator` is equal to the default constructed `integer_type`

Note

$0 + x = x + 0 = x$ and $0 - x = x - 0 = x$

Warning

it does **not** do the test for the neutral element to multiplication and division

Returns

`true` if it is the neutral element to addition and subtraction, `false` otherwise

```
4.9.4.9 template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed >
class CHKOP = NO_OPERATOR_CHECK> bool Commons::Math::Rational< T, GCD, CHKOP >::operator!=(
const Rational< T, GCD, CHKOP > & other ) const [inline]
```

test on inequality

Parameters

<code>in</code>	<code>other</code>	the Rational to test to
-----------------	--------------------	-------------------------

Returns

`true` if not equal, `false` otherwise

```
4.9.4.10 template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed >
class CHKOP = NO_OPERATOR_CHECK> Rational Commons::Math::Rational< T, GCD, CHKOP >::operator%(
const Rational< T, GCD, CHKOP > & other ) const [inline]
```

modulo a Rational

Parameters

<code>in</code>	<code>other</code>	the Rational to modulo
-----------------	--------------------	------------------------

Returns

a new Rational

```
4.9.4.11 template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename >
> class > class GCD, template< class, typename, bool > class CHKOP> Rational< T, GCD, CHKOP > &
Commons::Math::Rational< T, GCD, CHKOP >::operator%=( const Rational< T, GCD, CHKOP > & other )
```

modulo and assign a Rational

Parameters

<code>in</code>	<code>other</code>	the Rational to modulo and assign
-----------------	--------------------	-----------------------------------

Returns

the Rational

```
4.9.4.12 template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename  
> class > class GCD, template< class, typename, bool > class CHKOP> Rational< T, GCD, CHKOP >  
Commons::Math::Rational< T, GCD, CHKOP >::operator* ( const Rational< T, GCD, CHKOP > & other )  
const
```

multiply a Rational

Parameters

<i>in</i>	<i>other</i>	the Rational to multiply
-----------	--------------	--------------------------

Returns

a new Rational

4.9.4.13 `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Rational< T, GCD, CHKOP > & Commons::Math::Rational< T, GCD, CHKOP >::operator*=(const Rational< T, GCD, CHKOP > & other)`

multiply and assign a Rational

Parameters

<i>in</i>	<i>other</i>	the Rational to multiply and assign
-----------	--------------	-------------------------------------

Returns

the Rational

4.9.4.14 `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Rational< T, GCD, CHKOP > Commons::Math::Rational< T, GCD, CHKOP >::operator+(const Rational< T, GCD, CHKOP > & other) const`

add a Rational

Parameters

<i>in</i>	<i>other</i>	the Rational to add
-----------	--------------	---------------------

Returns

a new Rational

4.9.4.15 `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> Rational Commons::Math::Rational< T, GCD, CHKOP >::operator+() const [inline]`

get a copy of the Rational

Returns

a copy of Rational

4.9.4.16 `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> Rational& Commons::Math::Rational< T, GCD, CHKOP >::operator++() [inline]`

pre-increment a Rational

the result will be $\frac{numerator+denominator}{denominator} \Rightarrow numerator + 1$

Returns

the incremented Rational

```
4.9.4.17 template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed >
class CHKOP = NO_OPERATOR_CHECK> Rational Commons::Math::Rational< T, GCD, CHKOP >::operator++
( int ) [inline]
```

post-increment a Rational

See also

[operator++\(\)](#)

Returns

a copy of Rational

```
4.9.4.18 template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed
> class CHKOP = NO_OPERATOR_CHECK> Rational& Commons::Math::Rational< T, GCD, CHKOP
>::operator+=( const Rational< T, GCD, CHKOP > & other ) [inline]
```

add and assign a Rational

Parameters

<code>in</code>	<code>other</code>	the Rational to add and assign
-----------------	--------------------	--------------------------------

Returns

the Rational

```
4.9.4.19 template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed >
class CHKOP = NO_OPERATOR_CHECK> Rational Commons::Math::Rational< T, GCD, CHKOP >::operator- (
const Rational< T, GCD, CHKOP > & other ) const [inline]
```

subtract a Rational

Parameters

<code>in</code>	<code>other</code>	the Rational to subtract
-----------------	--------------------	--------------------------

Returns

a new Rational

```
4.9.4.20 template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed >
class CHKOP = NO_OPERATOR_CHECK> Rational Commons::Math::Rational< T, GCD, CHKOP >::operator- (
) const [inline]
```

get a negated copy of the Rational

Returns

a negated copy of Rational

```
4.9.4.21 template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed >
> class CHKOP = NO_OPERATOR_CHECK> Rational& Commons::Math::Rational< T, GCD, CHKOP
>::operator-- ( ) [inline]
```

pre-decrement a Rational

the result will be $\frac{\text{numerator} - \text{denominator}}{\text{denominator}} \Rightarrow \text{numerator} - 1$

Returns

the incremented Rational

```
4.9.4.22 template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed >
class CHKOP = NO_OPERATOR_CHECK> Rational Commons::Math::Rational< T, GCD, CHKOP >::operator--
( int ) [inline]
```

post-decrement a Rational

See also

[operator--\(\)](#)

Returns

a copy of Rational

```
4.9.4.23 template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed >
> class CHKOP = NO_OPERATOR_CHECK> Rational& Commons::Math::Rational< T, GCD, CHKOP
>::operator=( const Rational< T, GCD, CHKOP > & other ) [inline]
```

subtract and assign a Rational

Parameters

in	<i>other</i>	the Rational to subtract and assign
----	--------------	-------------------------------------

Returns

the Rational

```
4.9.4.24 template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed >
class CHKOP = NO_OPERATOR_CHECK> Rational Commons::Math::Rational< T, GCD, CHKOP >::operator/ (
const Rational< T, GCD, CHKOP > & other ) const [inline]
```

divide a Rational

Parameters

<i>in</i>	<i>other</i>	the Rational to divide
-----------	--------------	------------------------

Returns

a new Rational

```
4.9.4.25 template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed
> class CHKOP = NO_OPERATOR_CHECK> Rational& Commons::Math::Rational< T, GCD, CHKOP
>::operator/= ( const Rational< T, GCD, CHKOP > & other ) [inline]
```

divide and assign a Rational

Parameters

<i>in</i>	<i>other</i>	the Rational to divide and assign
-----------	--------------	-----------------------------------

Returns

the Rational

```
4.9.4.26 template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD, template< class, typename, bool > class CHKOP> bool Commons::Math::Rational< T,
GCD, CHKOP >::operator< ( const Rational< T, GCD, CHKOP > & other ) const
```

test if less than

Parameters

<i>in</i>	<i>other</i>	the Rational to test to
-----------	--------------	-------------------------

Returns

true if less than *other*, false otherwise

```
4.9.4.27 template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed >
class CHKOP = NO_OPERATOR_CHECK> bool Commons::Math::Rational< T, GCD, CHKOP >::operator<= (
const Rational< T, GCD, CHKOP > & other ) const [inline]
```

test if less or equal than

Parameters

<i>in</i>	<i>other</i>	the Rational to test to
-----------	--------------	-------------------------

Returns

true if less or equal than *other*, false otherwise

```
4.9.4.28 template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed
> class CHKOP = NO_OPERATOR_CHECK> Rational& Commons::Math::Rational< T, GCD, CHKOP
>::operator= ( const Rational< T, GCD, CHKOP > & another ) [inline]
```

assign another Rational

Parameters

<code>in</code>	<code>another</code>	the Rational to assign
-----------------	----------------------	------------------------

4.9.4.29 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> template<typename NumberType > Rational& Commons::Math::Rational< T, GCD, CHKOP >::operator= (const NumberType & number) [inline]`

assigns from a `NumberType`

The `number` is approximated to a `Rational`, then it gets assigned

See also

[Rational\(const NumberType &number\)](#)
[Commons::Math::EPSILON](#)

Template Parameters

<code>NumberType</code>	type of the number to approximate
-------------------------	-----------------------------------

Parameters

<code>in</code>	<code>number</code>	the number to assign
-----------------	---------------------	----------------------

4.9.4.30 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> bool Commons::Math::Rational< T, GCD, CHKOP >::operator== (const Rational< T, GCD, CHKOP > & other) const [inline]`

test on equality

Parameters

<code>in</code>	<code>other</code>	the Rational to test to
-----------------	--------------------	-------------------------

Returns

`true` if equal, `false` otherwise

4.9.4.31 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> bool Commons::Math::Rational< T, GCD, CHKOP >::operator> (const Rational< T, GCD, CHKOP > & other) const [inline]`

test if greater than

Parameters

<code>in</code>	<code>other</code>	the Rational to test to
-----------------	--------------------	-------------------------

Returns

`true` if greater than `other`, `false` otherwise


```
4.9.4.32 template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename >
class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed >
class CHKOP = NO_OPERATOR_CHECK> bool Commons::Math::Rational< T, GCD, CHKOP >::operator>= (
const Rational< T, GCD, CHKOP > & other ) const [inline]
```

test if greater or equal than

Parameters

<code>in</code>	<code>other</code>	the Rational to test to
-----------------	--------------------	-------------------------

Returns

`true` if greater or equal than `other`, `false` otherwise

4.9.4.33 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> std::string Commons::Math::Rational< T, GCD, CHKOP >::str(bool mixed = false) const`

generates the string representation of Rational

Parameters

<code>in</code>	<code>mixed</code>	if <code>true</code> , than a mixed (improper) fraction is generated
-----------------	--------------------	--

Returns

the string representation of Rational

4.9.5 Friends And Related Function Documentation

4.9.5.1 `template<typename NumberType, typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> bool operator!=(const NumberType & n, const Rational< T, GCD, CHKOP > & o)` [[related](#)]

Template Parameters

<code>NumberType</code>	the number type
<code>T</code>	the storage type
<code>GCD</code>	the GCD algorithm
<code>CHKOP</code>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.2 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType > bool operator!=(const Rational< T, GCD, CHKOP > & o, const NumberType & n)` [[related](#)]

Template Parameters

<code>T</code>	the storage type
<code>GCD</code>	the GCD algorithm
<code>NumberType</code>	the number type
<code>CHKOP</code>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.3 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType > Rational< T, GCD, CHKOP > operator%(const Rational< T, GCD, CHKOP > & o, const NumberType & n)` [[related](#)]

Template Parameters

<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>NumberType</i>	the number type
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.4 `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Rational< T, GCD, CHKOP > operator%(const NumberType & n, const Rational< T, GCD, CHKOP > & o)` [\[related\]](#)

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.5 `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> NumberType & operator%=(NumberType & n, const Rational< T, GCD, CHKOP > & o)` [\[related\]](#)

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.6 `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType > Rational< T, GCD, CHKOP > operator*(const Rational< T, GCD, CHKOP > & o, const NumberType & n)` [\[related\]](#)

Template Parameters

<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>NumberType</i>	the number type

<i>CHKOP</i>	checked operator
--------------	------------------

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.7 `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Rational< T, GCD, CHKOP > operator*(const NumberType & n, const Rational< T, GCD, CHKOP > & o)` [related]

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.8 `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> NumberType & operator*(NumberType & n, const Rational< T, GCD, CHKOP > & o)` [related]

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.9 `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType > Rational< T, GCD, CHKOP > operator+(const Rational< T, GCD, CHKOP > & o, const NumberType & n)` [related]

Template Parameters

<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>NumberType</i>	the number type
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.10 `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Rational< T, GCD, CHKOP > operator+(const NumberType & n, const Rational< T, GCD, CHKOP > & o)` [related]

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.11 `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> NumberType & operator+=(NumberType & n, const Rational< T, GCD, CHKOP > & o)` [\[related\]](#)

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.12 `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType > Rational< T, GCD, CHKOP > operator-(const Rational< T, GCD, CHKOP > & o, const NumberType & n)` [\[related\]](#)

Template Parameters

<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>NumberType</i>	the number type
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.13 `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Rational< T, GCD, CHKOP > operator-(const NumberType & n, const Rational< T, GCD, CHKOP > & o)` [\[related\]](#)

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm

<i>CHKOP</i>	checked operator
--------------	------------------

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.14 `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> NumberType & operator-= (NumberType & n, const Rational< T, GCD, CHKOP > & o)` [related]

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.15 `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType > Rational< T, GCD, CHKOP > operator/ (const Rational< T, GCD, CHKOP > & o, const NumberType & n)` [related]

Template Parameters

<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>NumberType</i>	the number type
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.16 `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> Rational< T, GCD, CHKOP > operator/ (const NumberType & n, const Rational< T, GCD, CHKOP > & o)` [related]

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.17 `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> NumberType & operator/= (NumberType & n, const Rational< T, GCD, CHKOP > & o)` [related]

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.18 `template<typename NumberType, typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> bool operator< (const NumberType & n, const Rational< T, GCD, CHKOP > & o)` [\[related\]](#)

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.19 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType > bool operator< (const Rational< T, GCD, CHKOP > & o, const NumberType & n)` [\[related\]](#)

Template Parameters

<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>NumberType</i>	the number type
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.20 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> std::ostream& operator<< (std::ostream & o, const Rational< T, GCD, CHKOP > & r)` [\[friend\]](#)

output stream operator

Sends a string representation of Rational to a `std::ostream`

See also

`str(bool)`

Parameters

out	<i>o</i>	the stream to send to
in	<i>r</i>	the Rational to send to the stream <i>o</i>

Returns

the stream *o*

4.9.5.21 `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> bool operator<= (const NumberType & n, const Rational< T, GCD, CHKOP > & o)` [related]

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.22 `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType > bool operator<= (const Rational< T, GCD, CHKOP > & o, const NumberType & n)` [related]

Template Parameters

<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>NumberType</i>	the number type
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.23 `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> bool operator== (const NumberType & n, const Rational< T, GCD, CHKOP > & o)` [related]

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.24 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType > bool operator==(const Rational< T, GCD, CHKOP > & o, const NumberType & n)` [\[related\]](#)

Template Parameters

<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>NumberType</i>	the number type
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.25 `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> bool operator> (const NumberType & n, const Rational< T, GCD, CHKOP > & o)` [related]

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.26 `template<typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType > bool operator> (const Rational< T, GCD, CHKOP > & o, const NumberType & n)` [related]

Template Parameters

<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>NumberType</i>	the number type
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.27 `template<typename NumberType , typename T , template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP> bool operator>= (const NumberType & n, const Rational< T, GCD, CHKOP > & o)` [related]

Template Parameters

<i>NumberType</i>	the number type
<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm

<i>CHKOP</i>	checked operator
--------------	------------------

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.28 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD, template< class, typename, bool > class CHKOP, typename NumberType > bool operator>= (const Rational< T, GCD, CHKOP > & o, const NumberType & n)` [[related](#)]

Template Parameters

<i>T</i>	the storage type
<i>GCD</i>	the GCD algorithm
<i>NumberType</i>	the number type
<i>CHKOP</i>	checked operator

See also

[ENABLE_OVERFLOW_CHECK](#)

4.9.5.29 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> std::istream& operator>> (std::istream & i, Rational< T, GCD, CHKOP > & r)` [[friend](#)]

reads in an expression from a `std::istream` and assign it to `r`

See also

[Rational\(const char *expr\)](#)

Parameters

<code>in</code>	<code>i</code>	the stream to read from
<code>out</code>	<code>r</code>	the Rational to assign to

Returns

the stream `i`

4.9.6 Member Data Documentation

4.9.6.1 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> const Rational< T, GCD, CHKOP >::integer_type Commons::Math::Rational< T, GCD, CHKOP >::one_ = integer_type(1)` [[static](#)]

represents *one* in the given [Rational::integer_type](#)

4.9.6.2 `template<typename T, template< typename, bool, template< class, typename, bool > class, template< typename > class > class GCD = GCD_euclid_fast, template< class, typename=T, bool=std::numeric_limits< T >::is_signed > class CHKOP = NO_OPERATOR_CHECK> const Rational< T, GCD, CHKOP >::integer_type Commons::Math::Rational< T, GCD, CHKOP >::zero_ = integer_type()` [[static](#)]

represents *zero* in the given [Rational::integer_type](#)

4.10 Commons::Math::RationalExpressionTraits< ExprT > Struct Template Reference

Traits struct for expression templates.

```
#include <expr_rational.h>
```

Public Types

- typedef ExprT [expr_type](#)
the deduced expression type
- typedef [expr_type](#) [literal_type](#)
the deduced literal type
- typedef [expr_type](#) [variable_type](#)
the deduced variable type

4.10.1 Detailed Description

```
template<class ExprT>struct Commons::Math::RationalExpressionTraits< ExprT >
```

Traits struct for expression templates.

This traits struct can be used to get the types for variable declarations.

Examples:

- to create a literal `l` :

```
const Commons::Math::RationalExpressionTraits
<Commons::Math::gmp_rational>::expr_type
&l ( Commons::Math::mk_rat_lit (
    Commons::Math::gmp_rational ( 1, 1 ) ) );
```

- to create a variable `x` using the prototype `l` :

```
const Commons::Math::RationalExpressionTraits
<Commons::Math::gmp_rational>::variable_type
&x ( Commons::Math::mk_rat_proto_var ( l ) );
```

See also

[Commons::Math::mk_rat_lit\(\)](#)
[Commons::Math::mk_rat_proto_var\(\)](#)

4.10.2 Member Typedef Documentation

4.10.2.1 `template<class ExprT > typedef ExprT Commons::Math::RationalExpressionTraits< ExprT >::expr_type`

the deduced expression type

4.10.2.2 `template<class ExprT > typedef expr_type Commons::Math::RationalExpressionTraits< ExprT >::literal_type`

the deduced literal type

```
4.10.2.3 template<class ExprT > typedef expr_type Commons::Math::RationalExpressionTraits< ExprT
>::variable_type
```

the deduced variable type

4.11 Commons::Math::TYPE_CONVERT< T > Struct Template Reference

Type conversion policy class.

```
#include <rational.h>
```

Public Member Functions

- [TYPE_CONVERT](#) (const T &v)
Constructs a type converter.
- `template<typename U >`
`U convert () const`
converts the value to U

4.11.1 Detailed Description

```
template<typename T>struct Commons::Math::TYPE_CONVERT< T >
```

Type conversion policy class.

Specialize this class to convert a type from T to U

Template Parameters

<i>T</i>	the type to convert from
----------	--------------------------

4.11.2 Constructor & Destructor Documentation

```
4.11.2.1 template<typename T > Commons::Math::TYPE_CONVERT< T >::TYPE_CONVERT ( const T & v )
[inline], [explicit]
```

Constructs a type converter.

Parameters

<i>in</i>	<i>v</i>	the value to convert
-----------	----------	----------------------

4.11.3 Member Function Documentation

```
4.11.3.1 template<typename T > template<typename U > U Commons::Math::TYPE_CONVERT< T >::convert ( )
const [inline]
```

converts the value to U

Template Parameters

<i>U</i>	the type to convert to
----------	------------------------

Index

- abs
 - Commons::Math::Rational, [32](#)
 - Expression templates, [9](#)
- CLN - Class Library for Numbers extensions, [5](#)
 - CLN_EPSILON, [6](#)
 - CLN_PRECISION, [6](#)
 - cln_rational, [6](#)
- CLN_EPSILON
 - CLN - Class Library for Numbers extensions, [6](#)
- CLN_PRECISION
 - CLN - Class Library for Numbers extensions, [6](#)
- cln_rational
 - CLN - Class Library for Numbers extensions, [6](#)
- Commons::Math::ENABLE_OVERFLOW_CHECK< Op, T, IsSigned >, [19](#)
- Commons::Math::EPSILON
 - value, [20](#)
- Commons::Math::EPSILON< T >, [19](#)
- Commons::Math::GCD_cln< T, IsSigned, CHKOP, C↔ONV >, [20](#)
- Commons::Math::GCD_euclid< T, IsSigned, CHKO↔P, CONV >, [20](#)
- Commons::Math::GCD_euclid_fast< T, IsSigned, CH↔KOP, CONV >, [21](#)
- Commons::Math::GCD_gmp< T, IsSigned, CHKOP, C↔ONV >, [21](#)
- Commons::Math::GCD_stein< T, IsSigned, CHKOP, CONV >, [22](#)
- Commons::Math::NO_OPERATOR_CHECK< Op, T, IsSigned >, [22](#)
- Commons::Math::Rational
 - abs, [32](#)
 - denominator, [32](#)
 - integer_type, [28](#)
 - inverse, [32](#)
 - invert, [32](#)
 - mod, [33](#)
 - mod_type, [28](#)
 - numerator, [33](#)
 - one_, [51](#)
 - op_divides, [28](#)
 - op_minus, [28](#)
 - op_modulus, [29](#)
 - op_multiplies, [29](#)
 - op_negate, [29](#)
 - op_plus, [29](#)
 - operator NumberType, [33](#)
 - operator!, [33](#)
 - operator!=, [34, 42](#)
 - operator<, [39, 47](#)
 - operator<<, [47](#)
 - operator<=, [39, 48](#)
 - operator>, [40, 50](#)
 - operator>>, [51](#)
 - operator>=, [40, 50, 51](#)
 - operator*, [34, 43, 44](#)
 - operator*=: [36, 44](#)
 - operator+, [36, 44](#)
 - operator++, [36, 37](#)
 - operator+=, [37, 45](#)
 - operator-, [37, 45](#)
 - operator--, [38](#)
 - operator-=, [38, 46](#)
 - operator/, [38, 46](#)
 - operator/=, [39, 46](#)
 - operator=, [39, 40](#)
 - operator==, [40, 48](#)
 - operator%, [34, 42, 43](#)
 - operator%=, [34, 43](#)
 - Rational, [29–31](#)
 - str, [42](#)
 - zero_, [51](#)
- Commons::Math::Rational< T, GCD, CHKOP >, [24](#)
- Commons::Math::RationalExpressionTraits
 - expr_type, [52](#)
 - literal_type, [52](#)
 - variable_type, [52](#)
- Commons::Math::RationalExpressionTraits< ExprT >, [52](#)
- Commons::Math::TYPE_CONVERT
 - convert, [53](#)
 - TYPE_CONVERT, [53](#)
- Commons::Math::TYPE_CONVERT< T >, [53](#)
- convert
 - Commons::Math::TYPE_CONVERT, [53](#)
- denominator
 - Commons::Math::Rational, [32](#)
- eval_rat_expr
 - Expression templates, [9](#)
- expr_type
 - Commons::Math::RationalExpressionTraits, [52](#)
- Expression templates, [7](#)
 - abs, [9](#)
 - eval_rat_expr, [9](#)
 - inv, [9, 10](#)
 - mk_rat_lit, [10](#)
 - mk_rat_proto_var, [10, 11](#)

- GMP_EPSILON
 - GNU Multiple Precisions extensions, 12
- GNU Multiple Precisions extensions, 12
 - GMP_EPSILON, 12
 - gmp_rational, 12
- gmp_rational
 - GNU Multiple Precisions extensions, 12
- Greatest common divisor algorithms, 17
- Inflnt extensions, 14
 - infint_rational, 14
- infint_rational
 - Inflnt extensions, 14
- integer_type
 - Commons::Math::Rational, 28
- inv
 - Expression templates, 9, 10
- inverse
 - Commons::Math::Rational, 32
- invert
 - Commons::Math::Rational, 32
- literal_type
 - Commons::Math::RationalExpressionTraits, 52
- mk_rat_lit
 - Expression templates, 10
- mk_rat_proto_var
 - Expression templates, 10, 11
- mod
 - Commons::Math::Rational, 33
- mod_type
 - Commons::Math::Rational, 28
- modf
 - Rational fraction class, 16
- numerator
 - Commons::Math::Rational, 33
- one_
 - Commons::Math::Rational, 51
- op_divides
 - Commons::Math::Rational, 28
- op_minus
 - Commons::Math::Rational, 28
- op_modulus
 - Commons::Math::Rational, 29
- op_multiplies
 - Commons::Math::Rational, 29
- op_negate
 - Commons::Math::Rational, 29
- op_plus
 - Commons::Math::Rational, 29
- operator NumberType
 - Commons::Math::Rational, 33
- operator!
 - Commons::Math::Rational, 33
- operator!=
 - Commons::Math::Rational, 34, 42
- operator<
 - Commons::Math::Rational, 39, 47
- operator<<
 - Commons::Math::Rational, 47
- operator<=
 - Commons::Math::Rational, 39, 48
- operator>
 - Commons::Math::Rational, 40, 50
- operator>>
 - Commons::Math::Rational, 51
- operator>=
 - Commons::Math::Rational, 40, 50, 51
- operator*
 - Commons::Math::Rational, 34, 43, 44
- operator*=
 - Commons::Math::Rational, 36, 44
- operator+
 - Commons::Math::Rational, 36, 44
- operator++
 - Commons::Math::Rational, 36, 37
- operator+=
 - Commons::Math::Rational, 37, 45
- operator-
 - Commons::Math::Rational, 37, 45
- operator--
 - Commons::Math::Rational, 38
- operator=
 - Commons::Math::Rational, 38, 46
- operator/
 - Commons::Math::Rational, 38, 46
- operator/=
 - Commons::Math::Rational, 39, 46
- operator=
 - Commons::Math::Rational, 39, 40
- operator==
 - Commons::Math::Rational, 40, 48
- operator%
 - Commons::Math::Rational, 34, 42, 43
- operator%=
 - Commons::Math::Rational, 34, 43
- Rational
 - Commons::Math::Rational, 29–31
- Rational fraction class, 15
 - modf, 16
- str
 - Commons::Math::Rational, 42
- TYPE_CONVERT
 - Commons::Math::TYPE_CONVERT, 53
- value
 - Commons::Math::EPSILON, 20
- variable_type
 - Commons::Math::RationalExpressionTraits, 52
- zero_
 - Commons::Math::Rational, 51