A Proposal[1] to add the Infinite Precision Integer and Rational to the C++ Standard Library

M.J. Kronenburg
e-mail: M.Kronenburg@inter.nl.net

1 November 2004

[1]Document number N1718=04-0138

Contents

List of Tables

Chapter 1

General

1.1 Motivation

The need of arithmetic types not fitting into the data width of the processor increases. For example on a 32-bit machine the typical ranges and precisions (excluding the sign) of the arithmetic base types are given in the table below:

Table 1.1: Range and precision of arithmetic base types

base type	**approximate range**	**approximate precision**
`int`	1 - 10^9	-
`float`	10^{-38} - 10^{38}	6 decimals
`double`	10^{-308} - 10^{308}	15 decimals
`long double`	10^{-4932} - 10^{4932}	19 decimals

For exceeding these ranges and precisions, classes can be created that combine many base type elements and a sign into arithmetic classes whose data sizes are only limited by available memory size. The mathematical operators and functions are overloaded.
Two arithmetic classes are proposed in this document for exceeding these ranges: the infinite precision integer and the rational. The rational is a combination of two infinite precision integers, the numerator and the denominator, and is relatively easy to implement on top of the infinite precision integer. For completeness, the arbitrary precision real is mentioned in the introducing chapters, but not proposed, because the mathematical functions of the arbitrary precision real are difficult to implement. The arbitrary precision real is also built on top of the infinite precision integer. When the infinite precision integer and the rational are accepted, the proposal for the arbitrary precision real may follow at a later stage.

1.2 Impact on the Standard

There is no impact of change on the standard, as the new arithmetic classes are fully self-contained with their own memory management. For input/output, the standard istream and ostream library classes are used, and internally many other standard library elements are used.

1.3 Existing Implementations

Currently a number of implementations of the infinite precision integer exist that give a good overview of design and performance issues:

1. The `Integer` class in the Gnu C++ library
   (C++, limited to about 10^5 decimals [7]).

2. The Gnu Multiple Precision Arithmetic Library
   (C with assembler, unlimited [6]).

3. The `Integer` class developed by myself.
   (C++ with assembler, unlimited)

Below these implementations are referred to as implementations 1, 2 and 3. There are also a few commercial computer algebra programs and libraries available.

1.4 Performance

1.4.1 Infinite Precision Integer Performance

The complexities in this document are provided as functions of N, which is the number of decimals or bits.

Table 1.2: Specific complexities and their meaning

complexity	**meaning**
N	number of decimals or bits
M(N)	multiplication of two infinite precision integers
D(N)	division or remainder of two infinite precision integers
G(N)	greatest common divisor of two infinite precision integers

On most systems M(N)<D(N)<G(N). In some cases the performance may not be of any interest, but users that start using the infinite precision integer and rational may tend to test the class for large arguments and compare results with the well known commercial computer algebra programs. This may be even more the case when on top of the infinite precision integer the arbitrary precision real is defined. Therefore the complexities of the basic operations should be carefully considered in the design and implementation.
The data granularity is the width of the data chunks that are operated on, which is not necessarily the processor data width. On 32-bit processors, implementation 1 uses short 16-bit granularity, because this means that all shift and carry operations can be done using 32-bit `ints` and no

assembler is needed. The performance of all arithmetic operations is much better using the granularity of the processor data width, which means that assembler is unavoidable.
For the multiplication of two large infinite precision integers, a number of algorithms exist (N is number of decimals or bits):

Table 1.3: Infinite precision integer multiplication algorithms

algorithm	**implementation**	**complexity**	**decimals**	**reference**
basecase	1,2,3	$O(N^2)$	$1 - 10^2$	
Karatsuba	2,3	$O(N^{1.585})$	$10^2 - 10^3$	[1,5]
3-way Toom-Cook	2	$O(N^{1.465})$	$> 10^3$	[1,5]
16-way Toom-Cook	3	$O(N^{1.239})$	$> 10^3$	[1]
Schönhage NTT	2	$O(N \log N \log\log N)$	$> 10^3$	[3,4,5]
Strassen FFT	-	$O(N \log^2 N)$	$> 10^3$	[1,4,6]

(NTT is Number Theoretic Transform, FFT is Fast Fourier Transform). The Strassen FFT algorithm uses floating-point arithmetic, which means that its accuracy cannot be mathematically guaranteed for very large arguments [5], and it is therefore not used in the quoted implementations, but its performance is the best of all. Implementation 1 only uses basecase multiplication, and therefore for large arguments its performance is poor. In implementation 3 I found that 16-way Toom-Cook is faster than Schönhage NTT (up to some very large argument not known to me).
For division and remainder recursive algorithms lead to better performance for large integers, and the same is true for the instream and outstream, which means conversion from binary to decimal notation and vice versa. For very large integers, division with Newton's method [1] becomes fastest, which is $O(M(N))$. For the greatest common divisor and extended greatest common divisor, Euclidean, binary and other algorithms exist, which are mostly $O(N^2)$ [1,2,5].

1.4.2 Rational Performance

The performance of the rational is mostly determined by the performance of the greatest common divisor and the multiplication of the infinite precision integer. After most of the arithmetic operations of the rational a normalization is necessary, which means a division of the numerator and denominator by their greatest common divisor, thus keeping the rational objects unique and of minimal size. Therefore these arithmetic operations have complexity $O(G(N))$. When the performance of the rational needs to be optimized, for computations where no boolean or stream operators are required, this normalization can be temporarily switched off by setting `rational::autonorm` to false, reducing the complexity of arithmetic operations with complexity $O(G(N))$ to complexity $O(M(N))$. In that case, as soon as boolean or stream operators are required, the objects must be explicitly normalized with `x.normalize()`. The automatic normalization can be switched on again by setting `rational::autonorm` to true, which is also the default. When performance of the rational is not an issue, the user is never required to use `rational::autonorm` or `x.normalize()`.

1.5 Unresolved Issues

1. The required complexities of `pow`, `powmod`, `fac`, `sqrt` and `random` are currently not clear.

2. The overall performance of the infinite precision integer should be comparable with the overall performance of such functionality in the well known commercial computer algebra programs or libraries. Existing implementations should therefore be tested on performance.

3. Avoiding temporaries in expressions (for example converting `a:=b+c;` into `a:=b; a+=c;`) can be done at compile time with templates, or (preferably) as a compiler code optimization flag. This should work for all arithmetic classes.

Chapter 2

Design Decisions

2.1 Classes

The arithmetic types of infinite precision integer, rational and arbitrary precision real are expressed as C++ classes. Each object of these classes contains the data that is unique and sufficient to represent its numerical value. The arithmetic operations are overloaded, which makes any expression possible that is also possible for the arithmetic base types.

Table 2.1: New arithmetic class types

arithmetic type	**meaning**	**class**
infinite precision integer	integer with infinite range	`integer`
rational	fraction of two infinite precision integers	`rational`
arbitrary precision real	real with arbitrary precision and range	`real`

The infinite precision integer class consists of two pointers to the begin and end of a contiguous memory block that contains its numerical absolute value, and a sign that can be 1 (positive), 0 (zero) or -1 (negative). An alternative would be to put the data in an STL vector, but as many infinite precision integer operations are performed at bit level, this would imply a dependency on the STL vector container implementation.

The use of C++ classes is much easier for the user that the use of a C-style interface, although in some cases a C-style interface may result in a slightly better performance. The use of a pure C++ interface is however recommended.

The rational class simply consists of two infinite precision integers: the numerator and denominator. After changing these values, they must always be normalized, that is divided by their greatest common divisor, so that the rational objects are unique and of minimal size. Given an infinite precision integer with greatest common divisor, the rational is relatively easy to implement.

For the rational, the possibility for using a template as `rational<int>` and `rational<integer>` is not recommended, because the range of the numerator and denominator of `rational<int>` would be only half the range of the base type `int`. This is because addition or subtraction of rationals imply multiplications of numerators and denominators. This would be difficult to explain to the user. The prevention of such overflow in a template is practically impossible. The

gain in performance by using `rational<int>` would be very limited. Therefore in this proposal no template is used for the rational, and the use in the rational of infinite precision integers for numerator and denominator is implied.
The arbitrary precision real consists of two infinite precision integers for the mantissa and the exponent, whose values are bounded by the range and the precision that can be set by the user. As mentioned the arbitrary precision real requires a separate proposal, but as the arbitrary precision real is built on top of the infinite precision integer, that proposal may refer to the contents of this proposal.

2.2 Interface

The interface to the arithmetic types is provided by specifying the possible operations on objects of the classes described in the previous sections. These operations are a list of all possible arithmetic, boolean, bitwise and other operators and functions. Given these operators and functions, listed in the following two chapters, different declarations in the header can be possible: an operator can be declared as member or non-member function, and an argument can be passed as a const or non-const parameter. Therefore the recommended declarations of the arithmetic types in the header file are provided (see chapter Proposed Headers).
Some arithmetic operators are preferable declared as non-member functions, because only in that case implicit conversion of the left-hand side argument is possible, as for example in `x = 3 + y;`, where the `int 3` is implicitly converted to the class type of `y` via the constructor from `int`. In the case of `x += 3;` defining a special `+=` operator for the `int` is a bit faster than implicit conversion, but given the complexity of the `+=` operator, this performance gain is in general very small. However implementations may add specialized operators to the interface. Constructors of the arithmetic classes from arithmetic base types and conversion functions from the arithmetic classes to the arithmetic base types are present, which may ease the use of these new arithmetic classes with existing software.
For the infinite precision integer, constructors from C-style and C++ strings are present. In this way, constants can be expressed as:

```
const integer x( "-12345678901234567890" );
```

The interface of the arithmetic class types should enable the use of templates for (for example) vectors and matrices, by using definitions that can both be used for arithmetic base types and for the new arithmetic class types. For example, when the boolean function even would have been defined as a member function `x.even()`, then a template using this function could never work for base type `int`. Therefore the even function is defined as a non-member function `even(x)`, so that a similar function can be defined for base type `int`. Some functions have both a member and a non-member variant, like `x.abs()` and `abs(x)`. This is because in this case the member variant has a better complexity, which makes the member variant preferable for non-template application.

2.3 Error Handling

When an error condition in the infinite precision integer or rational operations occurs, an exception of the appropriate exception class type is thrown. This exception class is derived from the `std::exception` class. It has a member method `appendCaller` that appends a string with caller information, so that the caller of the function that generated the exception, can catch it, append caller information to the exception, and throw it. The user that catches this exception in the main program and calls the exceptions `what()` function gets a string with a list of successive called functions that leads to the function that generated the exception. This way a maximum of exception information is provided to the user.
The internal type of exception is one of an `enum` list, which can be division by zero, memory allocation etc. This `enum` list is preferred in favour of multiple derivation of the exception class, as long as each arithmetic class derives its own exception class from `std::exception`.
The infinite precision integer has a global variable maxbits that gives the maximum number of bits. When an infinite precision integer exceeds this maximum, an exception is generated. On most systems the default value of maxbits may be set to a value where the maximum memory size is exceeded, or it may be set to a lower value by the user for debugging purposes. When maxbits is set to zero, the maximum number of bits is infinite.
The table below gives an overview of possible error conditions.

Table 2.2: Error conditions

error condition	**meaning**	**typical operations**
error_unknown	unknown error (default)	all
error_bitoverflow	number of bits > maxbits	arithmetic, left shifts, bit operations
error_iszero	integer is zero error	highestbit, lowestbit
error_isnegative	integer is negative error	arithmetic sqrt, pow, powmod
error_divbyzero	division by zero error	arithmetic division and remainder
error_memalloc	memory allocation error	all
error_conversion	base type conversion overflow error	conversions to base types
error_basefield	input base conversion error	instream

Chapter 3

Infinite Precision Integer

3.1 Constructors

The constructor of the infinite precision integer can take as argument variables of any arithmetic base type or a string type. The values of the floating point base types are truncated towards zero. The destructor has complexity O(1).

Table 3.1: Infinite precision integer constructors requirements

expression	**return type**	**pre/post-condition**	**complexity**
`integer()`	`integer`	returns an `integer` with the value 0	O(1)
`integer(ivar)`	`integer`	returns an `integer` with the value of the `int` variable `ivar`	O(1)
`integer(uivar)`	`integer`	returns an `integer` with the value of the `unsigned int` variable `uivar`	O(1)
`integer(fvar)`	`integer`	returns an `integer` with the truncated value of the `float` variable `fvar`	O(1)
`integer(dvar)`	`integer`	returns an `integer` with the truncated value of the `double` variable `dvar`	O(1)
`integer(ldvar)`	`integer`	returns an `integer` with the truncated value of the `long double` variable `ldvar`	O(1)
`integer(csvar)`	`integer`	returns an `integer` with the decimal value of the C-string variable `csvar`	O($< N^2$)
`integer(strvar)`	`integer`	returns an `integer` with the decimal value of the `string` variable `strvar`	O($< N^2$)
`integer(iivar)`	`integer`	returns an `integer` with the value of the `integer` variable `iivar`	O(N)

3.2 Operators

3.2.1 Arithmetic Operators

Table 3.2: Infinite precision integer arithmetic operators requirements

expression	**return type**	**pre/post-condition**	**complexity**
`x = y`	`integer` reference	`integer x` is assigned by `integer y`	O(N)
`++x`	`integer` reference	`integer x` is incremented by one	amortized O(1)
`--x`	`integer` reference	`integer x` is decremented by one	amortized O(1)
`x++`	`integer`	`integer x` is incremented by one and the original value is returned	O(N)
`x--`	`integer`	`integer x` is decremented by one and the original value is returned	O(N)
`-x`	`integer`	returns the negated `integer x`	O(N)
`x += y`	`integer` reference	`integer x` is added by `integer y`	O(N)
`x -= y`	`integer` reference	`integer x` is subtracted by `integer y`	O(N)
`x *= y`	`integer` reference	`integer x` is multiplied by `integer y`	M(N)=O($< N^2$)
`x /= y`	`integer` reference	`integer x` is divided by `integer y`	D(N)=O($< N^2$)
`x %= y`	`integer` reference	`integer x` is divided as remainder by `integer y`	D(N)
`x + y`	`integer`	returns the sum of `integers x` and `y`	O(N)
`x - y`	`integer`	returns the difference of `integers x` and `y`	O(N)
`x * y`	`integer`	returns the product of `integers x` and `y`	M(N)
`x / y`	`integer`	returns the quotient of `integers x` and `y`	D(N)
`x % y`	`integer`	returns the remainder of `integers x` and `y`	D(N)

3.2.2 Boolean Operators

Table 3.3: Infinite precision integer boolean operators requirements

expression	return type	pre/post-condition	complexity
`x == y`	`bool`	returns true if `integer x` is equal to `integer y`, otherwise false	O(N)
`x != y`	`bool`	returns true if `integer x` is not equal to `integer y`, otherwise false	O(N)
`x > y`	`bool`	returns true if `integer x` is greater than `integer y`, otherwise false	O(N)
`x >= y`	`bool`	returns true if `integer x` is greater than or equal to `integer y`, otherwise false	O(N)
`x < y`	`bool`	returns true if `integer x` is less than `integer y`, otherwise false	O(N)
`x <= y`	`bool`	returns true if `integer x` is less than or equal to `integer y`, otherwise false	O(N)

3.2.3 Bitwise Operators

Table 3.4: Infinite precision integer bitwise operators requirements

<table>
<tr><th>expression</th><th>return type</th><th>pre/post-condition</th><th>complexity</th></tr>
<tr><td><code>x |= y</code></td><td><code>integer</code> reference</td><td><code>integer x</code> is or-ed with <code>integer y</code></td><td>O(N)</td></tr>
<tr><td><code>x &= y</code></td><td><code>integer</code> reference</td><td><code>integer x</code> is and-ed with <code>integer y</code></td><td>O(N)</td></tr>
<tr><td><code>x ^= y</code></td><td><code>integer</code> reference</td><td><code>integer x</code> is xor-ed with <code>integer y</code></td><td>O(N)</td></tr>
<tr><td><code>x | y</code></td><td><code>integer</code></td><td>returns <code>integer x</code> or-ed with <code>integer y</code></td><td>O(N)</td></tr>
<tr><td><code>x & y</code></td><td><code>integer</code></td><td>returns <code>integer x</code> and-ed with <code>integer y</code></td><td>O(N)</td></tr>
<tr><td><code>x ^ y</code></td><td><code>integer</code></td><td>returns <code>integer x</code> xor-ed with <code>integer y</code></td><td>O(N)</td></tr>
</table>

3.2.4 Shift Operators

When a left shift results in an infinite precision integer with more bits than `integer::maxbits` (see section 3.3.2), an exception is thrown.

Table 3.5: Infinite precision integer shift operators requirements

expression	return type	pre/post-condition	complexity
`x <<= iivar`	`integer` reference	`integer x` is left shifted by the `integer iivar`	$O(N)$
`x >>= iivar`	`integer` reference	`integer x` is right shifted by the `integer iivar`	$O(N)$
`x << iivar`	`integer`	returns `integer x` left shifted by the `integer iivar`	$O(N)$
`x >> iivar`	`integer`	returns `integer x` right shifted by the `integer iivar`	$O(N)$

3.2.5 Stream Operators

The numerical notation of the infinite precision integer is determined by the basefield flag of the corresponding stream, which can be set with the stream manipulators `std::dec`, `std::hex` and `std::oct`.

Table 3.6: Infinite precision integer stream operators requirements

expression	return type	pre/post-condition	complexity
`is >> x`	`istream` reference	`integer x` is read from the `istream` is	$O(< N^2)$
`os << x`	`ostream` reference	`integer x` is written to the `ostream` os	$O(< N^2)$

3.3 Functions

3.3.1 Arithmetic Functions

The greatest common divisor of two infinite precision integers is the greatest integer that divides both values. This function is also needed for the normalization of rationals (see section 4.3.1).

Table 3.7: Infinite precision integer arithmetic functions requirements

expression	return type	pre/post-condition	complexity
`abs(x)`	`integer`	returns the absolute value of `integer x`	O(N)
`sqr(x)`	`integer`	returns the square of `integer x`	M(N)
`sqrt(x)`	`integer`	returns the floor of the square root of `integer x`	?
`divmod(x,y,q,r)`	`void`	`integer q` becomes the quotient of `integer x` and `y`, and `integer r` becomes the remainder	D(N)
`pow(x,y)`	`integer`	returns the power of `integer x` by `integer y`	?
`powmod(x,y,z)`	`integer`	returns the power of `integer x` by `integer y`, modulo `integer z`	?
`fac(x)`	`integer`	returns the factorial of `integer x`	?
`random(x,y)`	`integer`	returns a random `integer` >= `integer x` and < `integer y`	?
`gcd(x,y)`	`integer`	returns the greatest common divisor of `integer x` and `integer y`	G(N)=O(N^2)
`lcm(x,y)`	`integer`	returns the least common multiple of `integer x` and `integer y`	G(N)
`extgcd(x,y,a,b)`	`integer`	returns `gcd(x,y)`, and `integers a` and `b` fulfill `xa + yb = gcd(x,y)`	O(N^2)

3.3.2 Member Functions

The default value of `integer::maxbits` may be related to the maximum available memory size. It may be set to a lower value by the user for debugging purposes. When `integer::maxbits` is zero, the maximum number of bits is infinite.

Table 3.8: Infinite precision integer member functions requirements

expression	return type	pre/post-condition	complexity
`integer::maxbits`	`integer`	the maximum number of bits of an `integer`	O(1)
`x.negate()`	`integer` reference	the sign of `integer x` is negated	O(1)
`x.abs()`	`integer` reference	the sign of `integer x` becomes positive	O(1)

3.3.3 Bit Functions

The bit numbering is such that the lowest bit has bit number 0. When the second parameter of `getbit`, `setbit` or `clearbit` is larger than `integer::maxbits` (see section 3.3.2), an exception is thrown.

Table 3.9: Infinite precision integer bit functions requirements

expression	return type	pre/post-condition	complexity
`getbit(x,iivar)`	`bool`	returns true if the bit with bit number `integer iivar` of `integer x` is 1, otherwise false	O(1)
`setbit(x,iivar)`	`void`	`integer x` has the bit with bit number `integer iivar` set to 1	O(1)
`clearbit(x,iivar)`	`void`	`integer x` has the bit with bit number `integer iivar` set to 0	O(1)
`lowestbit(x)`	`integer`	returns bit number of the lowest bit that is 1 of `integer x`	amortized O(1)
`highestbit(x)`	`integer`	returns bit number of the highest bit that is 1 of `integer x`	O(N)

3.3.4 Miscelaneous Functions

For conversion of infinite precision integers to floating point base types, truncation takes place towards zero.

Table 3.10: Infinite precision integer miscelaneous functions requirements

expression	return type	pre/post-condition	complexity
`sign(x)`	`int`	returns 0 if `integer x` is 0, 1 if `x` is greater than 0, and -1 if `x` is less than 0	O(1)
`even(x)`	`bool`	returns true if `integer x` is even, otherwise false	O(1)
`odd(x)`	`bool`	returns true if `integer x` is odd, otherwise false	O(1)
`swap(x,y)`	`void`	swaps the values of `integer x` and `integer y`	O(1)
`toint(x)`	`int`	returns an `int` with the value of `integer x`, if `x` is too large an exception is thrown	O(1)
`tofloat(x)`	`float`	returns a `float` with the truncated value of `integer x`, if `x` is too large an exception is thrown	O(1)
`todouble(x)`	`double`	returns a `double` with the truncated value of `integer x`, if `x` is too large an exception is thrown	O(1)
`tolongdouble(x)`	`long double`	returns a `long double` with the truncated value of `integer x`, if `x` is too large an exception is thrown	O(1)

Chapter 4

Rational

4.1 Constructors

The constructor of rational can be called with variables of any arithmetic base type or strings; they are implicitly converted to infinite precision integer type by the corresponding infinite precision integer constructor (see section 3.1). The sign of the resulting rational is the product of the signs of the first and second argument. When `rational::autonorm` (see section 4.3.1) is true, the rational is normalized. The destructor has complexity O(1).

Table 4.1: Rational constructors requirements

expression	**return type**	**pre/post-condition**	**complexity**
`rational()`	`rational`	returns a `rational` with the value 0	O(1)
`rational(iivar)`	`rational`	returns a `rational` with a numerator value of the `integer` variable `iivar` and a denominator value 1	O(1)
`rational(iivar1,iivar2)`	`rational`	returns a `rational` with a numerator value of the `integer` variable `iivar1` and a denominator value of the `integer` variable `iivar2`	O(1)
`rational(rvar)`	`rational`	returns a `rational` with the value of the `rational` variable `rvar`	O(N)

4.2 Operators

4.2.1 Arithmetic Operators

When `rational::autonorm` (see section 4.3.1) is true, the rational is normalized.
When `rational::autonorm` is false, the arithmetic functions with complexity O(G(N)) get complexity O(M(N)).

Table 4.2: Rational arithmetic operators requirements

expression	return type	pre/post-condition	complexity
`x = y`	`rational` reference	`rational x` is assigned by `rational y`	O(N)
`-x`	`rational`	returns the negated `rational x`	O(N)
`x += y`	`rational` reference	`rational x` is added by `rational y`	O(G(N))
`x -= y`	`rational` reference	`rational x` is subtracted by `rational y`	O(G(N))
`x *= y`	`rational` reference	`rational x` is multiplied by `rational y`	O(G(N))
`x /= y`	`rational` reference	`rational x` is divided by `rational y`	O(G(N))
`x + y`	`rational`	returns the sum of `rationals` `x` and `y`	O(G(N))
`x - y`	`rational`	returns the difference of `rationals` `x` and `y`	O(G(N))
`x * y`	`rational`	returns the product of `rationals` `x` and `y`	O(G(N))
`x / y`	`rational`	returns the quotient of `rationals` `x` and `y`	O(G(N))

4.2.2 Boolean Operators

Table 4.3: Rational boolean operators requirements

expression	return type	pre/post-condition	complexity
`x == y`	`bool`	returns true if `rational x` is equal to `rational y`, otherwise false	$O(N)$
`x != y`	`bool`	returns true if `rational x` is not equal to `rational y`, otherwise false	$O(N)$
`x > y`	`bool`	returns true if `rational x` is greater than `rational y`, otherwise false	$O(N)$
`x >= y`	`bool`	returns true if `rational x` is greater than or equal to `rational y`, otherwise false	$O(N)$
`x < y`	`bool`	returns true if `rational x` is less than `rational y`, otherwise false	$O(N)$
`x <= y`	`bool`	returns true if `rational x` is less than or equal to `rational y`, otherwise false	$O(N)$

4.2.3 Stream Operators

The notation of a rational is equal to the notation of the infinite precision integer numerator and denominator (see section 3.2.5) separated by the character /.

Table 4.4: Rational stream operators requirements

expression	return type	pre/post-condition	complexity
`is >> x`	`istream` reference	`rational x` is read from the `istream` is	$O(< N^2)$
`os << x`	`ostream` reference	`rational x` is written to the `ostream` os	$O(< N^2)$

4.3 Functions

4.3.1 Member Functions

Normalization of `rational` objects means that after each arithmetic operation the numerator and denominator are divided by their greatest common divisor, keeping the objects unique and of minimal size. For performance optimization, when no boolean or stream operators are needed, the normalization can be temporarily switched off by setting `rational::autonorm` to false, making the arithmetic operations that are of complexity O(G(N)) temporarily of complexity O(M(N)). In that case `x.normalize()` must be used to explicitly normalize a rational before calling boolean or stream operators. The automatic normalization can be switched on again by setting `rational::autonorm` to true, which is also the default.

Table 4.5: Rational member functions requirements

expression	**return type**	**pre/post-condition**	**complexity**
`rational::autonorm`	`bool`	when true (which is default), `rational` objects are normalized after each arithmetic operation; when false, they must be explicitly normalized	O(1)
`x.numerator()`	`integer` reference	returns the numerator of `rational x`	O(1)
`x.denominator()`	`integer` reference	returns the denominator of `rational x`	O(1)
`x.normalize()`	`rational` reference	`rational x` is normalized	O(1)
`x.negate()`	`rational` reference	the sign of `rational x` is negated	O(1)
`x.abs()`	`rational` reference	the sign of `rational x` becomes positive	O(1)
`x.invert()`	`rational` reference	`rational x` is inverted	O(1)
`x.trunc()`	`rational` reference	`rational x` is truncated towards zero	O(D(N))
`x.fract()`	`rational` reference	`rational x` becomes the remainder of truncation	O(D(N))

4.3.2 Arithmetic Functions

When `rational::autonorm` (see section 4.3.1) is true, the rational is normalized.
When `rational::autonorm` is false, the arithmetic functions with complexity O(G(N)) get complexity O(M(N)).

Table 4.6: Rational arithmetic functions requirements

expression	return type	pre/post-condition	complexity
`abs(x)`	`rational`	returns the absolute value of `rational x`	O(N)
`sqr(x)`	`rational`	returns the square of `rational x`	O(G(N))
`pow(x,y)`	`rational`	returns the exponent of `rational x` by `integer y`	?

4.3.3 Miscelaneous Functions

The conversion of rationals to floating point base types takes place by separate conversion of the infinite precision integer numerator and denominator (see section 3.3.4), and a floating point division.

Table 4.7: Rational miscelaneous functions requirements

expression	return type	pre/post-condition	complexity
`sign(x)`	`int`	returns 0 if `rational x` is 0, 1 if `x` is greater than 0, and -1 if `x` is less than 0	O(1)
`swap(x,y)`	`void`	swaps the values of `rational x` and `rational y`	O(1)
`tofloat(x)`	`float`	returns a `float` with the value of `rational x`, if `x` is too large an exception is thrown	O(1)
`todouble(x)`	`double`	returns a `double` with the value of `rational x`, if `x` is too large an exception is thrown	O(1)
`tolongdouble(x)`	`long double`	returns a `long double` with the value of `rational x`, if `x` is too large an exception is thrown	O(1)

Chapter 5

Proposed Headers

5.1 Infinite Precision Integer

```
class integer
{
 private:
  unsigned int *data, *maxdata;   // For exposition only
  signed char  thesign;           // For exposition only
 public:
  integer();
  integer( int );
  integer( unsigned int );
  integer( float );
  integer( double );
  integer( long double );
  integer( const char * );
  integer( const string & );
  integer( const integer & );
  ~integer();

  static integer maxbits;
  integer &negate();
  integer &abs();
  integer &operator=( const integer & );
  integer &operator++();
  integer &operator--();
  const integer operator++( int );
  const integer operator--( int );
  const integer operator-() const;
  integer &operator+=( const integer & );
  integer &operator-=( const integer & );
  integer &operator*=( const integer & );
  integer &operator/=( const integer & );
  integer &operator%=( const integer & );
  integer &operator|=( const integer & );
  integer &operator&=( const integer & );
  integer &operator^=( const integer & );
```

```
  integer &operator<<=( const integer & );
  integer &operator>>=( const integer & );
};

const integer operator+( const integer &, const integer & );
const integer operator-( const integer &, const integer & );
const integer operator*( const integer &, const integer & );
const integer operator/( const integer &, const integer & );
const integer operator%( const integer &, const integer & );

const bool operator==( const integer &, const integer & );
const bool operator!=( const integer &, const integer & );
const bool operator>( const integer &, const integer & );
const bool operator>=( const integer &, const integer & );
const bool operator<( const integer &, const integer & );
const bool operator<=( const integer &, const integer & );

const integer operator|( const integer &, const integer & );
const integer operator&( const integer &, const integer & );
const integer operator^( const integer &, const integer & );

const integer operator<<( const integer &, const integer & );
const integer operator>>( const integer &, const integer & );

ostream & operator<<( ostream &, const integer & );
istream & operator>>( istream &, integer & );

const integer abs( const integer & );
const integer sqr( const integer & );
const integer sqrt( const integer & );
void divmod( const integer &, const integer &, integer &, integer & );
const integer pow( const integer &, const integer & );
const integer powmod( const integer &, const integer &, const integer & );
const integer fac( const integer & );
const integer random( const integer &, const integer & );
const integer gcd( const integer &, const integer & );
const integer lcm( const integer &, const integer & );
const integer extgcd( const integer &, const integer &, integer &, integer & );

const bool getbit( const integer &, const integer & );
void setbit( integer &, const integer & );
void clearbit( integer &, const integer & );
const integer lowestbit( const integer & );
const integer highestbit( const integer & );
```

```
const int  sign( const integer & );
const bool even( const integer & );
const bool odd( const integer & );
void swap( integer &, integer & );

const int toint( const integer & );
const float tofloat( const integer & );
const double todouble( const integer & );
const long double tolongdouble( const integer & );
```

5.2 Rational

```
class rational
{
 private:
  integer numerator, denominator;   // For exposition only
 public:
  rational();
  rational( const integer & );
  rational( const integer &, const integer & );
  rational( const rational & );
  ~rational();

  static bool autonorm;
  integer &numerator();
  integer &denominator();
  rational &normalize();
  rational &negate();
  rational &abs();
  rational &invert();
  rational &trunc();
  rational &fract();
  rational &operator=( const rational & );
  const rational operator-() const;
  rational &operator+=( const rational & );
  rational &operator-=( const rational & );
  rational &operator*=( const rational & );
  rational &operator/=( const rational & );
};

const rational operator+( const rational &, const rational & );
const rational operator-( const rational &, const rational & );
const rational operator*( const rational &, const rational & );
```

```
const rational operator/( const rational &, const rational & );

const bool operator==( const rational &, const rational & );
const bool operator!=( const rational &, const rational & );
const bool operator>( const rational &, const rational & );
const bool operator>=( const rational &, const rational & );
const bool operator<( const rational &, const rational & );
const bool operator<=( const rational &, const rational & );

ostream & operator<<( ostream &, const rational & );
istream & operator>>( istream &, rational & );

const rational abs( const rational & );
const rational sqr( const rational & );
const rational pow( const rational &, const integer & );
const int sign( const rational & );
void swap( rational &, rational & );

const float tofloat( const rational & );
const double todouble( const rational & );
const long double tolongdouble( const rational & );
```

5.3 Error Handling

```
class numeric_exception : public std::exception
{ public:
    enum type_of_error {
      error_unknown, error_divbyzero, error_memalloc, ...
    };
    numeric_exception( type_of_error = error_unknown, ... );
    appendCaller( const string & );
    virtual const char *what() const;
  private:
    type_of_error error_type;
    string error_description;
};

class integer_exception : public numeric_exception
{  integer_exception( type_of_error );
   virtual const char *what() const;
};
```

```
class rational_exception : public numeric_exception
{  rational_exception( type_of_error );
   virtual const char *what() const;
};
```

Chapter 6

References

1. D.E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms (1998).

2. E. Bach and J. Shallit, Algorithmic Number Theory, Volume 1: Efficient Algorithms (1996).

3. P. Zimmermann, An implementation of Schönhage's multiplication algorithm (1992).

4. A. Schönhage and V. Strassen, Computing 7 (1971) 281.

5. Free Software Foundation, Gnu MP manual ed. 4.1.2 (2002).

6. http://numbers.computation.free.fr/Constants/Algorithms/fft.html

7. http://www.swox.com/gmp

8. http://www.math.utah.edu/docs/info/libg++_20.html