Boost.Multiprecision

John Maddock

Christopher Kormanyos

Table of Contents

Introduction

The Multiprecision Library provides integer, rational and floating-point types in C++ that have more range and precision than C++'s ordinary built-in types. The big number types in Multiprecision can be used with a wide selection of basic mathematical operations, elementary transcendental functions as well as the functions in Boost.Math. The Multiprecision types can also interoperate with the built-in types in C++ using clearly defined conversion rules. This allows Boost.Multiprecision to be used for all kinds of mathematical calculations involving integer, rational and floating-point types requiring extended range and precision.

Multiprecision consists of a generic interface to the mathematics of large numbers as well as a selection of big number back ends, with support for integer, rational and floating-point types. Boost.Multiprecision provides a selection of back ends provided off-the-rack in including interfaces to GMP, MPFR, MPIR, TomMath as well as its own collection of Boost-licensed, header-only back ends for integers, rationals and floats. In addition, user-defined back ends can be created and used with the interface of Multiprecision, provided the class implementation adheres to the necessary concepts.

Depending upon the number type, precision may be arbitrarily large (limited only by available memory), fixed at compile time (for example 50 or 100 decimal digits), or a variable controlled at run-time by member functions. The types are expression-template-enabled for better performance than naive user-defined types.

The Multiprecision library comes in two distinct parts:

- An expression-template-enabled front-end `number` that handles all the operator overloading, expression evaluation optimization, and code reduction.

- A selection of back-ends that implement the actual arithmetic operations, and need conform only to the reduced interface requirements of the front-end.

Separation of front-end and back-end allows use of highly refined, but restricted license libraries where possible, but provides Boost license alternatives for users who must have a portable unconstrained license. Which is to say some back-ends rely on 3rd party libraries, but a header-only Boost license version is always available (if somewhat slower).

Should you just wish to cut to the chase and use a fully Boost-licensed number type, then skip to cpp_int for multiprecision integers, cpp_dec_float for multiprecision floating point types and cpp_rational for rational types.

The library is often used via one of the predefined typedefs: for example if you wanted an arbitrary precision integer type using GMP as the underlying implementation then you could use:

```
#include <boost/multiprecision/gmp.hpp>  // Defines the wrappers around the GMP library's types

boost::multiprecision::mpz_int myint;    // Arbitrary precision integer type.
```

Alternatively, you can compose your own multiprecision type, by combining `number` with one of the predefined back-end types. For example, suppose you wanted a 300 decimal digit floating-point type based on the MPFR library. In this case, there's no predefined typedef with that level of precision, so instead we compose our own:

```
#include <boost/multiprecision/mpfr.hpp>  // Defines the Backend type that wraps MPFR

namespace mp = boost::multiprecision;      // Reduce the typing a bit later...

typedef mp::number<mp::mpfr_float_backend<300> >  my_float;

my_float a, b, c; // These variables have 300 decimal digits precision
```

We can repeat the above example, but with the expression templates disabled (for faster compile times, but slower runtimes) by passing a second template argument to `number`:

```
#include <boost/multiprecision/mpfr.hpp>   // Defines the Backend type that wraps MPFR

namespace mp = boost::multiprecision;      // Reduce the typing a bit later...

typedef mp::number<mp::mpfr_float_backend<300>, et_off>  my_float;

my_float a, b, c; // These variables have 300 decimal digits precision
```

We can also mix arithmetic operations between different types, provided there is an unambiguous implicit conversion from one type to the other:

```
#include <boost/multiprecision/cpp_int.hpp>

namespace mp = boost::multiprecision;       // Reduce the typing a bit later...

mp::int128_t a(3), b(4);
mp::int512_t c(50), d;

d = c * a;    // OK, result of mixed arithmetic is an int512_t
```

Conversions are also allowed:

```
d = a; // OK, widening conversion.
d = a * b;  // OK, can convert from an expression template too.
```

However conversions that are inherently lossy are either declared explicit or else forbidden altogether:

```
d = 3.14;   // Error implicit conversion from float not allowed.
d = static_cast<mp::int512_t>(3.14);  // OK explicit construction is allowed
```

Mixed arithmetic will fail if the conversion is either ambiguous or explicit:

```
number<cpp_int_backend<>, et_off> a(2);
number<cpp_int_backend<>, et_on>  b(3);

b = a * b; // Error, implicit conversion could go either way.
b = a * 3.14; // Error, no operator overload if the conversion would be explicit.
```

Move Semantics

On compilers that support rvalue-references, class `number` is move-enabled if the underlying backend is.

In addition the non-expression template operator overloads (see below) are move aware and have overloads that look something like:

```
template <class B>
number<B, et_off> operator + (number&& a, const number& b)
{
    return std::move(a += b);
}
```

These operator overloads ensure that many expressions can be evaluated without actually generating any temporaries. However, there are still many simple expressions such as:

```
a = b * c;
```

Which don't noticeably benefit from move support. Therefore, optimal performance comes from having both move-support, and expression templates enabled.

Note that while "moved-from" objects are left in a sane state, they have an unspecified value, and the only permitted operations on them are destruction or the assignment of a new value. Any other operation should be considered a programming error and all of our backends will trigger an assertion if any other operation is attempted. This behavior allows for optimal performance on move-construction (i.e. no allocation required, we just take ownership of the existing object's internal state), while maintaining usability in the standard library containers.

Expression Templates

Class `number` is expression-template-enabled: that means that rather than having a multiplication operator that looks like this:

```
template <class Backend>
number<Backend> operator * (const number<Backend>& a, const number<Backend>& b)
{
   number<Backend> result(a);
   result *= b;
   return result;
}
```

Instead the operator looks more like this:

```
template <class Backend>
unmentionable-type operator * (const number<Backend>& a, const number<Backend>& b);
```

Where the "unmentionable" return type is an implementation detail that, rather than containing the result of the multiplication, contains instructions on how to compute the result. In effect it's just a pair of references to the arguments of the function, plus some compile-time information that stores what the operation is.

The great advantage of this method is the *elimination of temporaries*: for example the "naive" implementation of `operator*` above, requires one temporary for computing the result, and at least another one to return it. It's true that sometimes this overhead can be reduced by using move-semantics, but it can't be eliminated completely. For example, lets suppose we're evaluating a polynomial via Horner's method, something like this:

```
T a[7] = { /* some values */ };
//....
y = ((((((a[6] * x + a[5]) * x + a[4]) * x + a[3]) * x + a[2]) * x + a[1]) * x + a[0];
```

If type `T` is a `number`, then this expression is evaluated *without creating a single temporary value*. In contrast, if we were using the mpfr_class C++ wrapper for MPFR - then this expression would result in no less than 11 temporaries (this is true even though mpfr_class does use expression templates to reduce the number of temporaries somewhat). Had we used an even simpler wrapper around MPFR like mpreal things would have been even worse and no less that 24 temporaries are created for this simple expression (note - we actually measure the number of memory allocations performed rather than the number of temporaries directly, note also that the mpf_class wrapper that will be supplied with GMP-5.1 reduces the number of temporaries to pretty much zero). Note that if we compile with expression templates disabled and rvalue-reference support on, then actually still have no wasted memory allocations as even though temporaries are created, their contents are moved rather than copied. [1]

---

[1] The actual number generated will depend on the compiler, how well it optimises the code, and whether it supports rvalue references. The number of 11 temporaries was generated with Visual C++ 10

> ## ⚠ Important
>
> Expression templates can radically reorder the operations in an expression, for example:
>
> a = (b * c) * a;
>
> Will get transformed into:
>
> a *= c; a *= b;
>
> If this is likely to be an issue for a particular application, then they should be disabled.

This library also extends expression template support to standard library functions like `abs` or `sin` with `number` arguments. This means that an expression such as:

```
y = abs(x);
```

can be evaluated without a single temporary being calculated. Even expressions like:

```
y = sin(x);
```

get this treatment, so that variable 'y' is used as "working storage" within the implementation of `sin`, thus reducing the number of temporaries used by one. Of course, should you write:

```
x = sin(x);
```

Then we clearly can't use `x` as working storage during the calculation, so then a temporary variable is created in this case.

Given the comments above, you might be forgiven for thinking that expression-templates are some kind of universal-panacea: sadly though, all tricks like this have their downsides. For one thing, expression template libraries like this one, tend to be slower to compile than their simpler cousins, they're also harder to debug (should you actually want to step through our code!), and rely on compiler optimizations being turned on to give really good performance. Also, since the return type from expressions involving `numbers` is an "unmentionable implementation detail", you have to be careful to cast the result of an expression to the actual number type when passing an expression to a template function. For example, given:

```
template <class T>
void my_proc(const T&);
```

Then calling:

```
my_proc(a+b);
```

Will very likely result in obscure error messages inside the body of `my_proc` - since we've passed it an expression template type, and not a number type. Instead we probably need:

```
my_proc(my_number_type(a+b));
```

Having said that, these situations don't occur that often - or indeed not at all for non-template functions. In addition, all the functions in the Boost.Math library will automatically convert expression-template arguments to the underlying number type without you having to do anything, so:

```
mpfr_float_100 a(20), delta(0.125);
boost::math::gamma_p(a, a + delta);
```

Will work just fine, with the `a + delta` expression template argument getting converted to an `mpfr_float_100` internally by the Boost.Math library.

One other potential pitfall that's only possible in C++11: you should never store an expression template using:

```
auto my_expression = a + b - c;
```

unless you're absolutely sure that the lifetimes of `a`, `b` and `c` will outlive that of `my_expression`.

And finally... the performance improvements from an expression template library like this are often not as dramatic as the reduction in number of temporaries would suggest. For example if we compare this library with mpfr_class and mpreal, with all three using the underlying MPFR library at 50 decimal digits precision then we see the following typical results for polynomial execution:

Table 1. Evaluation of Order 6 Polynomial.

Library	Relative Time	Relative number of memory allocations
number	1.0 (0.00957s)	1.0 (2996 total)
mpfr_class	1.1 (0.0102s)	4.3 (12976 total)
mpreal	1.6 (0.0151s)	9.3 (27947 total)

As you can see, the execution time increases a lot more slowly than the number of memory allocations. There are a number of reasons for this:

• The cost of extended-precision multiplication and division is so great, that the times taken for these tend to swamp everything else.

• The cost of an in-place multiplication (using `operator*=`) tends to be more than an out-of-place `operator*` (typically `operator*=` has to create a temporary workspace to carry out the multiplication, where as `operator*` can use the target variable as workspace). Since the expression templates carry out their magic by converting out-of-place operators to in-place ones, we necessarily take this hit. Even so the transformation is more efficient than creating the extra temporary variable, just not by as much as one would hope.

Finally, note that `number` takes a second template argument, which, when set to `et_off` disables all the expression template machinery. The result is much faster to compile, but slower at runtime.

We'll conclude this section by providing some more performance comparisons between these three libraries, again, all are using MPFR to carry out the underlying arithmetic, and all are operating at the same precision (50 decimal digits):

Table 2. Evaluation of Boost.Math's Bessel function test data

Library	Relative Time	Relative Number of Memory Allocations
mpfr_float_50	1.0 (5.78s)	1.0 (1611963)
number<mpfr_float_backend<50>, et_off> (but with rvalue reference support)	1.1 (6.29s)	2.64 (4260868)
mpfr_class	1.1 (6.28s)	2.45 (3948316)
mpreal	1.65 (9.54s)	8.21 (13226029)

Table 3. Evaluation of Boost.Math's Non-Central T distribution test data

Library	Relative Time	Relative Number of Memory Allocations
number	1.0 (263s)	1.0 (127710873)
number<mpfr_float_backend<50>, et_off> (but with rvalue reference support)	1.0 (260s)	1.2 (156797871)
mpfr_class	1.1 (287s)	2.1 (268336640)
mpreal	1.5 (389s)	3.6 (466960653)

The above results were generated on Win32 compiling with Visual C++ 2010, all optimizations on (/Ox), with MPFR 3.0 and MPIR 2.3.0.

Tutorial

In order to use this library you need to make two choices:

- What kind of number do I want (integer, floating point or rational).

- Which back-end do I want to perform the actual arithmetic (Boost-supplied, GMP, MPFR, Tommath etc)?

Integer Types

The following back-ends provide integer arithmetic:

Backend Type	Header	Radix	Dependencies	Pros	Cons
cpp_int	boost/multiprecision/cpp_int.hpp	2	None	Very versatile, Boost licensed, all C++ integer type which support both arbitrary precision and fixed precision integer types.	Slower than GMP, though typically not as slow as libtommath
gmp_int	boost/multiprecision/gmp.hpp	2	GMP	Very fast and efficient back-end.	Dependency on GNU licensed GMP library.
tom_int	boost/multiprecision/tommath.hpp	2	libtommath	Public domain back-end with no licence restrictions.	Slower than GMP.

cpp_int

```
#include <boost/multiprecision/cpp_int.hpp>
```

```
namespace boost{ namespace multiprecision{

typedef unspecified-type limb_type;

enum cpp_integer_type    { signed_magnitude, unsigned_magnitude };
enum cpp_int_check_type  { checked, unchecked };

template <unsigned MinDigits = 0,
          unsigned MaxDits = 0,
          cpp_integer_type SignType = signed_magnitude,
          cpp_int_check_type Checked = unchecked,
          class Allocator = std::allocator<limb_type> >
class cpp_int_backend;
//
// Expression templates default to et_off if there is no allocator:
//
template <unsigned MinDigits, unsigned MaxDigits, cpp_integer_type Sign↵
Type, cpp_int_check_type Checked>
struct expression_template_default<cpp_int_backend<MinDigits, MaxDigits, SignType, Checked, void> >
{ static const expression_template_option value = et_off; };

typedef number<cpp_int_backend<> >               cpp_int;     // arbitrary precision integer
typedef rational_adaptor<cpp_int_backend<> >     cpp_rational_backend;
typedef number<cpp_rational_backend>             cpp_rational; // arbitrary precision rational ↵
number

// Fixed precision unsigned types:
typedef number<cpp_int_backend<128, 128, unsigned_magnitude, unchecked, void> >   uint128_t;
typedef number<cpp_int_backend<256, 256, unsigned_magnitude, unchecked, void> >   uint256_t;
typedef number<cpp_int_backend<512, 512, unsigned_magnitude, unchecked, void> >   uint512_t;
typedef number<cpp_int_backend<1024, 1024, unsigned_magnitude, unchecked, void> > uint1024_t;

// Fixed precision signed types:
typedef number<cpp_int_backend<128, 128, signed_magnitude, unchecked, void> >    int128_t;
typedef number<cpp_int_backend<256, 256, signed_magnitude, unchecked, void> >    int256_t;
typedef number<cpp_int_backend<512, 512, signed_magnitude, unchecked, void> >    int512_t;
typedef number<cpp_int_backend<1024, 1024, signed_magnitude, unchecked, void> >  int1024_t;

// Over again, but with checking enabled this time:
typedef number<cpp_int_backend<0, 0, signed_magnitude, checked> >                checked_cpp_int;
typedef rational_adaptor<cpp_int_backend<0, 0, signed_magnitude, checked> >      checked_cpp_ra↵
tional_backend;
typedef number<cpp_rational_backend>                                            checked_cpp_ra↵
tional;

// Checked fixed precision unsigned types:
typedef number<cpp_int_backend<128, 128, unsigned_magnitude, checked, void> >    ↵
 checked_uint128_t;
typedef number<cpp_int_backend<256, 256, unsigned_magnitude, checked, void> >    ↵
 checked_uint256_t;
typedef number<cpp_int_backend<512, 512, unsigned_magnitude, checked, void> >    ↵
 checked_uint512_t;
typedef number<cpp_int_backend<1024, 1024, unsigned_magnitude, checked, void> >  ↵
 checked_uint1024_t;

// Fixed precision signed types:
typedef number<cpp_int_backend<128, 128, signed_magnitude, checked, void> >      checked_int128_t;
typedef number<cpp_int_backend<256, 256, signed_magnitude, checked, void> >      checked_int256_t;
typedef number<cpp_int_backend<512, 512, signed_magnitude, checked, void> >      checked_int512_t;
typedef number<cpp_int_backend<1024, 1024, signed_magnitude, checked, void> >    ↵
 checked_int1024_t;

}} // namespaces
```

The `cpp_int_backend` type is normally used via one of the convenience typedefs given above.

This back-end is the "Swiss Army Knife" of integer types as it can represent both fixed and arbitrary precision integer types, and both signed and unsigned types. There are five template arguments:

MinBits   Determines the number of Bits to store directly within the object before resorting to dynamic memory allocation. When zero, this field is determined automatically based on how many bits can be stored in union with the dynamic storage header: setting a larger value may improve performance as larger integer values will be stored internally before memory allocation is required.

MaxBits   Determines the maximum number of bits to be stored in the type: resulting in a fixed precision type. When this value is the same as MinBits, then the Allocator parameter is ignored, as no dynamic memory allocation will ever be performed: in this situation the Allocator parameter should be set to type `void`. Note that this parameter should not be used simply to prevent large memory allocations, not only is that role better performed by the allocator, but fixed precision integers have a tendency to allocate all of MaxBits of storage more often than one would expect.

SignType   Determines whether the resulting type is signed or not. Note that for arbitrary precision types this parameter must be `signed_magnitude`. For fixed precision types then this type may be either `signed_magnitude` or `unsigned_magnitude`.

Checked   This parameter has two values: `checked` or `unchecked`. See below.

Allocator   The allocator to use for dynamic memory allocation, or type `void` if MaxBits == MinBits.

When the template parameter Checked is set to `checked` then the result is a *checked-integer*, checked and unchecked integers have the following properties:

Condition	Checked-Integer	Unchecked-Integer
Numeric overflow in fixed precision arithmetic	Throws a `std::overflow_error`.	Performs arithmetic modulo $2^{MaxBits}$
Constructing an integer from a value that can not be represented in the target type	Throws a `std::range_error`.	Converts the value modulo $2^{MaxBits}$, signed to unsigned conversions extract the last MaxBits bits of the 2's complement representation of the input value.
Unsigned subtraction yielding a negative value.	Throws a `std::range_error`.	Yields the value that would result from treating the unsigned type as a 2's complement signed type.
Attempting a bitwise operation on a negative value.	Throws a `std::range_error`	Yields the value, but not the bit pattern, that would result from performing the operation on a 2's complement integer type.

Things you should know when using this type:

- Default constructed `cpp_int_backends` have the value zero.

- Division by zero results in a `std::overflow_error` being thrown.

- Construction from a string that contains invalid non-numeric characters results in a `std::runtime_error` being thrown.

- Since the precision of `cpp_int_backend` is necessarily limited when the allocator parameter is void, care should be taken to avoid numeric overflow when using this type unless you actually want modulo-arithmetic behavior.

- The type uses a sign-magnitude representation internally, so type `int128_t` has 128-bits of precision plus an extra sign bit. In this respect the behaviour of these types differs from built-in 2's complement types. In might be tempting to use a 127-bit type

instead, and indeed this does work, but behaviour is still slightly different from a 2's complement built-in type as the min and max values are identical (apart from the sign), where as they differ by one for a true 2's complement type. That said it should be noted that there's no requirement for built-in types to be 2's complement either - it's simply that this is the most common format by far.

- Attempting to print negative values as either an Octal or Hexadecimal string results in a `std::runtime_error` being thrown, this is a direct consequence of the sign-magnitude representation.

- The fixed precision types `[checked_][u]intXXX_t` have expression template support turned off - it seems to make little difference to the performance of these types either way - so we may as well have the faster compile times by turning the feature off.

- Unsigned types support subtraction - the result is "as if" a 2's complement operation had been performed as long as they are not *checked-integers* (see above). In other words they behave pretty much as a built in integer type would in this situation. So for example if we were using `uint128_t` then `uint128_t(1)-4` would result in the value `0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD` of type `uint128_t`. However, had this operation been performed on `checked_uint128_t` then a `std::range_error` would have been thrown.

- Unary negation of unsigned types results in a compiler error (static assertion).

- This backend supports rvalue-references and is move-aware, making instantiations of `number` on this backend move aware.

- When used at fixed precision, the size of this type is always one machine word larger than you would expect for an N-bit integer: the extra word stores both the sign, and how many machine words in the integer are actually in use. The latter is an optimisation for larger fixed precision integers, so that a 1024-bit integer has almost the same performance characteristics as a 128-bit integer, rather than being 4 times slower for addition and 16 times slower for multiplication (assuming the values involved would always fit in 128 bits). Typically this means you can use an integer type wide enough for the "worst case scenario" with only minor performance degradation even if most of the time the arithmetic could in fact be done with a narrower type.

- When used at fixed precision and MaxBits is smaller than the number of bits in the largest native integer type, then internally `cpp_int_backend` switches to a "trivial" implementation where it is just a thin wrapper around a single integer. Note that it will still be slightly slower than a bare native integer, as it emulates a signed-magnitude representation rather than simply using the platforms native sign representation: this ensures there is no step change in behavior as a cpp_int grows in size.

- Fixed precision `cpp_int`'s have some support for `constexpr` values and user-defined literals, see here for the full description. For example `0xfffff_cppi1024` specifies a 1024-bit integer with the value 0xffff. This can be used to generate compile time constants that are too large to fit into any built in number type.

Example:

```cpp
#include <boost/multiprecision/cpp_int.hpp>

using namespace boost::multiprecision;

int128_t v = 1;

// Do some fixed precision arithmetic:
for(unsigned i = 1; i <= 20; ++i)
   v *= i;

std::cout << v << std::endl; // prints 20!

// Repeat at arbitrary precision:
cpp_int u = 1;
for(unsigned i = 1; i <= 100; ++i)
   u *= i;

std::cout << u << std::endl; // prints 100!
```

gmp_int

```cpp
#include <boost/multiprecision/gmp.hpp>
```

```
namespace boost{ namespace multiprecision{

class gmp_int;

typedef number<gmp_int >          mpz_int;

}} // namespaces
```

The `gmp_int` back-end is used via the typedef `boost::multiprecision::mpz_int`. It acts as a thin wrapper around the GMP `mpz_t` to provide an integer type that is a drop-in replacement for the native C++ integer types, but with unlimited precision.

As well as the usual conversions from arithmetic and string types, type `mpz_int` is copy constructible and assignable from:

• The GMP native types: `mpf_t`, `mpz_t`, `mpq_t`.

• Instances of `number<T>` that are wrappers around those types: `number<gmp_float<N> >`, `number<gmp_rational>`.

It's also possible to access the underlying `mpz_t` via the `data()` member function of `gmp_int`.

Things you should know when using this type:

• No changes are made to the GMP library's global settings - so you can safely mix this type with existing code that uses GMP.

• Default constructed `gmp_ints` have the value zero (this is GMP's default behavior).

• Formatted IO for this type does not support octal or hexadecimal notation for negative values, as a result performing formatted output on this type when the argument is negative and either of the flags `std::ios_base::oct` or `std::ios_base::hex` are set, will result in a `std::runtime_error` will be thrown.

• Conversion from a string results in a `std::runtime_error` being thrown if the string can not be interpreted as a valid integer.

• Division by zero results in a `std::overflow_error` being thrown.

• Although this type is a wrapper around GMP it will work equally well with MPIR. Indeed use of MPIR is recommended on Win32.

• This backend supports rvalue-references and is move-aware, making instantiations of `number` on this backend move aware.

Example:

```
#include <boost/multiprecision/gmp.hpp>

using namespace boost::multiprecision;

mpz_int v = 1;

// Do some arithmetic:
for(unsigned i = 1; i <= 1000; ++i)
   v *= i;

std::cout << v << std::endl; // prints 1000!

// Access the underlying representation:
mpz_t z;
mpz_init(z);
mpz_set(z, v.backend().data());
```

tom_int

```
#include <boost/multiprecision/tommath.hpp>
```

```
namespace boost{ namespace multiprecision{

class tommath_int;

typedef number<tommath_int >        tom_int;

}} // namespaces
```

The `tommath_int` back-end is used via the typedef `boost::multiprecision::tom_int`. It acts as a thin wrapper around the libtommath `tom_int` to provide an integer type that is a drop-in replacement for the native C++ integer types, but with unlimited precision.

Things you should know when using this type:

- Default constructed objects have the value zero (this is libtommath's default behavior).

- Although `tom_int` is mostly a drop in replacement for the builtin integer types, it should be noted that it is a rather strange beast as it's a signed type that is not a 2's complement type. As a result the bitwise operations `|` `&` `^` will throw a `std::runtime_error` exception if either of the arguments is negative. Similarly the complement operator~ is deliberately not implemented for this type.

- Formatted IO for this type does not support octal or hexadecimal notation for negative values, as a result performing formatted output on this type when the argument is negative and either of the flags `std::ios_base::oct` or `std::ios_base::hex` are set, will result in a `std::runtime_error` will be thrown.

- Conversion from a string results in a `std::runtime_error` being thrown if the string can not be interpreted as a valid integer.

- Division by zero results in a `std::overflow_error` being thrown.

Example:

```
#include <boost/multiprecision/tommath.hpp>

boost::multiprecision::tom_int v = 1;

// Do some arithmetic:
for(unsigned i = 1; i <= 1000; ++i)
   v *= i;

std::cout << v << std::endl; // prints 1000!
std::cout << std::hex << v << std::endl; // prints 1000! in hex format

try{
   std::cout << std::hex << -v << std::endl; // Ooops! can't print a negative value in hex format!
}
catch(const std::runtime_error& e)
{
   std::cout << e.what() << std::endl;
}

try{
   // v is not a 2's complement type, bitwise operations are only supported
   // on positive values:
   v = -v & 2;
}
catch(const std::runtime_error& e)
{
   std::cout << e.what() << std::endl;
}
```

Examples

Factorials

In this simple example, we'll write a routine to print out all of the factorials which will fit into a 128-bit integer. At the end of the routine we do some fancy iostream formatting of the results:

```
#include <boost/multiprecision/cpp_int.hpp>
#include <iostream>
#include <iomanip>
#include <vector>


void print_factorials()
{
   using boost::multiprecision::cpp_int;
   //
   // Print all the factorials that will fit inside a 128-bit integer.
   //
   // Begin by building a big table of factorials, once we know just how
   // large the largest is, we'll be able to "pretty format" the results.
   //
   // Calculate the largest number that will fit inside 128 bits, we could
   // also have used numeric_limits<int128_t>::max() for this value:
   cpp_int limit = (cpp_int(1) << 128) - 1;
   //
   // Our table of values:
   std::vector<cpp_int> results;
   //
   // Initial values:
   unsigned i = 1;
   cpp_int factorial = 1;
   //
   // Cycle through the factorials till we reach the limit:
   while(factorial < limit)
   {
      results.push_back(factorial);
      ++i;
      factorial *= i;
   }
   //
   // Lets see how many digits the largest factorial was:
   unsigned digits = results.back().str().size();
   //
   // Now print them out, using right justification, while we're at it
   // we'll indicate the limit of each integer type, so begin by defining
   // the limits for 16, 32, 64 etc bit integers:
   cpp_int limits[] = {
      (cpp_int(1) << 16) - 1,
      (cpp_int(1) << 32) - 1,
      (cpp_int(1) << 64) - 1,
      (cpp_int(1) << 128) - 1,
   };
   std::string bit_counts[] = { "16", "32", "64", "128" };
   unsigned current_limit = 0;
   for(unsigned j = 0; j < results.size(); ++j)
   {
      if(limits[current_limit] < results[j])
      {
         std::string message = "Limit of " + bit_counts[current_limit] + " bit integers";
        std::cout << std::setfill('.') << std::setw(digits+1) << std::right << message << std::set↵
```

```
fill(' ') << std::endl;
         ++current_limit;
      }
      std::cout << std::setw(digits + 1) << std::right << results[j] << std::endl;
   }
}
```

The output from this routine is:

```
                                               1
                                               2
                                               6
                                              24
                                             120
                                             720
                                            5040
                                           40320
...............Limit of 16 bit integers
                                          362880
                                         3628800
                                        39916800
                                       479001600
...............Limit of 32 bit integers
                                      6227020800
                                     87178291200
                                   1307674368000
                                  20922789888000
                                 355687428096000
                                6402373705728000
                              121645100408832000
                             2432902008176640000
...............Limit of 64 bit integers
                            51090942171709440000
                          1124000727777607680000
                         25852016738884976640000
                        620448401733239439360000
                      15511210043330985984000000
                     403291461126605635584000000
                   10888869450418352160768000000
                  304888344611713860501504000000
                 8841761993739701954543616000000
               265252859812191058636308480000000
              8222838654177922817725562880000000
            263130836933693530167218012160000000
          8683317618811886495518194401280000000
        295232799039604140847618609643520000000
```

Bit Operations

In this example we'll show how individual bits within an integer may be manipulated, we'll start with an often needed calculation of $2^n - 1$, which we could obviously implement like this:

```
using boost::multiprecision::cpp_int;

cpp_int b1(unsigned n)
{
   cpp_int r(1);
   return (r << n) - 1;
}
```

Calling:

```
std::cout << std::hex << std::showbase << b1(200) << std::endl;
```

Yields as expected:

```
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

However, we could equally just set the n'th bit in the result, like this:

```
cpp_int b2(unsigned n)
{
   cpp_int r(0);
   return --bit_set(r, n);
}
```

Note how the `bit_set` function sets the specified bit in its argument and then returns a reference to the result - which we can then simply decrement. The result from a call to `b2` is the same as that to `b1`.

We can equally test bits, so for example the n'th bit of the result returned from `b2` shouldn't be set unless we increment it first:

```
assert(!bit_test(b1(200), 200));      // OK
assert(bit_test(++b1(200), 200));     // OK
```

And of course if we flip the n'th bit after increment, then we should get back to zero:

```
assert(!bit_flip(++b1(200), 200));    // OK
```

Floating Point Numbers

The following back-ends provide floating point arithmetic:

Backend Type	Header	Radix	Dependencies	Pros	Cons
`cpp_dec_float<N>`	boost/multipreci-sion/cpp_dec_float.hpp	10	None	Header only, all C++ implementation. Boost licence.	Approximately 2x slower than the MPFR or GMP libraries.
`mpf_float<N>`	boost/multipreci-sion/gmp.hpp	2	GMP	Very fast and efficient back-end.	Dependency on GNU licensed GMP library.
`mpfr_float<N>`	boost/multipreci-sion/mpfr.hpp	2	GMP and MPFR	Very fast and efficient back-end, with its own standard library implementation.	Dependency on GNU licensed GMP and MPFR libraries.
`float128`	boost/multipreci-sion/float128.hpp	2	Either libquadmath or the Intel C++ Math library.	Very fast and efficient back-end for 128-bit floating point values (113-bit mantissa, equivalent to FORTRAN's QUAD real)	Depends on the compiler being either recent GCC or Intel C++ versions.

cpp_dec_float

```
#include <boost/multiprecision/cpp_dec_float.hpp>
```

```cpp
namespace boost{ namespace multiprecision{

template <unsigned Digits10, class ExponentType = boost::int32_t, class Allocator = void>
class cpp_dec_float;

typedef number<cpp_dec_float<50> > cpp_dec_float_50;
typedef number<cpp_dec_float<100> > cpp_dec_float_100;

}} // namespaces
```

The `cpp_dec_float` back-end is used in conjunction with `number`: It acts as an entirely C++ (header only and dependency free) floating-point number type that is a drop-in replacement for the native C++ floating-point types, but with much greater precision.

Type `cpp_dec_float` can be used at fixed precision by specifying a non-zero `Digits10` template parameter. The typedefs `cpp_dec_float_50` and `cpp_dec_float_100` provide arithmetic types at 50 and 100 decimal digits precision respectively. Optionally, you can specify an integer type to use for the exponent, this defaults to a 32-bit integer type which is more than large enough for the vast majority of use cases, but larger types such as `long long` can also be specified if you need a truly huge exponent range. In any case the ExponentType must be a built in signed integer type at least 2 bytes and 16-bits wide.

Normally `cpp_dec_float` allocates no memory: all of the space required for its digits are allocated directly within the class. As a result care should be taken not to use the class with too high a digit count as stack space requirements can grow out of control. If that represents a problem then providing an allocator as the final template parameter causes `cpp_dec_float` to dynamically allocate the memory it needs: this significantly reduces the size of `cpp_dec_float` and increases the viable upper limit on the number of digits at the expense of performance. However, please bear in mind that arithmetic operations rapidly become *very* expensive as the digit count grows: the current implementation really isn't optimized or designed for large digit counts.

There is full standard library and `numeric_limits` support available for this type.

Things you should know when using this type:

- Default constructed `cpp_dec_floats` have a value of zero.

- The radix of this type is 10. As a result it can behave subtly differently from base-2 types.

- The type has a number of internal guard digits over and above those specified in the template argument. Normally these should not be visible to the user.

- The type supports both infinities and NaN's. An infinity is generated whenever the result would overflow, and a NaN is generated for any mathematically undefined operation.

- There is a `std::numeric_limits` specialisation for this type.

- Any `number` instantiated on this type, is convertible to any other `number` instantiated on this type - for example you can convert from `number<cpp_dec_float<50> >` to `number<cpp_dec_float<SomeOtherValue> >`. Narrowing conversions are truncating and `explicit`.

- Conversion from a string results in a `std::runtime_error` being thrown if the string can not be interpreted as a valid floating point number.

- The actual precision of a `cpp_dec_float` is always slightly higher than the number of digits specified in the template parameter, actually how much higher is an implementation detail but is always at least 8 decimal digits.

- Operations involving `cpp_dec_float` are always truncating. However, note that since their are guard digits in effect, in practice this has no real impact on accuracy for most use cases.

cpp_dec_float example:

```cpp
#include <boost/multiprecision/cpp_dec_float.hpp>

using namespace boost::multiprecision;

// Operations at fixed precision and full numeric_limits support:
cpp_dec_float_100 b = 2;
std::cout << std::numeric_limits<cpp_dec_float_100>::digits << std::endl;
// Note that digits10 is the same as digits, since we're base 10! :
std::cout << std::numeric_limits<cpp_dec_float_100>::digits10 << std::endl;
// We can use any C++ std lib function, lets print all the digits as well:
std::cout << std::setprecision(std::numeric_limits<cpp_dec_float_100>::max_digits10)
   << log(b) << std::endl; // print log(2)
// We can also use any function from Boost.Math:
std::cout << boost::math::tgamma(b) << std::endl;
// These even work when the argument is an expression template:
std::cout << boost::math::tgamma(b * b) << std::endl;
// And since we have an extended exponent range we can generate some really large
// numbers here (4.0238726007709377354370243e+2564):
std::cout << boost::math::tgamma(cpp_dec_float_100(1000)) << std::endl;
```

gmp_float

```cpp
#include <boost/multiprecision/gmp.hpp>
```

```
namespace boost{ namespace multiprecision{

template <unsigned Digits10>
class gmp_float;

typedef number<gmp_float<50> >    mpf_float_50;
typedef number<gmp_float<100> >   mpf_float_100;
typedef number<gmp_float<500> >   mpf_float_500;
typedef number<gmp_float<1000> >  mpf_float_1000;
typedef number<gmp_float<0> >     mpf_float;

}} // namespaces
```

The `gmp_float` back-end is used in conjunction with `number` : it acts as a thin wrapper around the GMP `mpf_t` to provide an real-number type that is a drop-in replacement for the native C++ floating-point types, but with much greater precision.

Type `gmp_float` can be used at fixed precision by specifying a non-zero `Digits10` template parameter, or at variable precision by setting the template argument to zero. The typedefs mpf_float_50, mpf_float_100, mpf_float_500, mpf_float_1000 provide arithmetic types at 50, 100, 500 and 1000 decimal digits precision respectively. The typedef mpf_float provides a variable precision type whose precision can be controlled via the `numbers` member functions.

> **Note**
>
> This type only provides standard library and `numeric_limits` support when the precision is fixed at compile time.

As well as the usual conversions from arithmetic and string types, instances of `number<mpf_float<N> >` are copy constructible and assignable from:

- The GMP native types `mpf_t`, `mpz_t`, `mpq_t`.

- The `number` wrappers around those types: `number<mpf_float<M> >`, `number<gmp_int>`, `number<gmp_rational>`.

It's also possible to access the underlying `mpf_t` via the `data()` member function of `gmp_float`.

Things you should know when using this type:

- Default constructed `gmp_floats` have the value zero (this is the GMP library's default behavior).

- No changes are made to the GMP library's global settings, so this type can be safely mixed with existing GMP code.

- This backend supports rvalue-references and is move-aware, making instantiations of `number` on this backend move aware.

- It is not possible to round-trip objects of this type to and from a string and get back exactly the same value. This appears to be a limitation of GMP.

- Since the underlying GMP types have no notion of infinities or NaN's, care should be taken to avoid numeric overflow or division by zero. That latter will result in a std::overflow_error being thrown, while generating excessively large exponents may result in instability of the underlying GMP library (in testing, converting a number with an excessively large or small exponent to a string caused GMP to segfault).

- This type can equally be used with MPIR as the underlying implementation - indeed that is the recommended option on Win32.

- Conversion from a string results in a `std::runtime_error` being thrown if the string can not be interpreted as a valid floating point number.

- Division by zero results in a `std::overflow_error` being thrown.

GMP example:

```cpp
#include <boost/multiprecision/gmp.hpp>

using namespace boost::multiprecision;

// Operations at variable precision and limited standard library support:
mpf_float a = 2;
mpf_float::default_precision(1000);
std::cout << mpf_float::default_precision() << std::endl;
std::cout << sqrt(a) << std::endl; // print root-2

// Operations at fixed precision and full standard library support:
mpf_float_100 b = 2;
std::cout << std::numeric_limits<mpf_float_100>::digits << std::endl;
// We can use any C++ std lib function:
std::cout << log(b) << std::endl; // print log(2)
// We can also use any function from Boost.Math:
std::cout << boost::math::tgamma(b) << std::endl;
// These even work when the argument is an expression template:
std::cout << boost::math::tgamma(b * b) << std::endl;

// Access the underlying representation:
mpf_t f;
mpf_init(f);
mpf_set(f, a.backend().data());
```

mpfr_float

```cpp
#include <boost/multiprecision/mpfr.hpp>
```

```cpp
namespace boost{ namespace multiprecision{

enum mpfr_allocation_type
{
   allocate_stack,
   allocate_dynamic
};

template <unsigned Digits10, mpfr_allocation_type AllocateType = allocate_dynamic>
class mpfr_float_backend;

typedef number<mpfr_float_backend<50> >    mpfr_float_50;
typedef number<mpfr_float_backend<100> >   mpfr_float_100;
typedef number<mpfr_float_backend<500> >   mpfr_float_500;
typedef number<mpfr_float_backend<1000> >  mpfr_float_1000;
typedef number<mpfr_float_backend<0> >     mpfr_float;

typedef number<mpfr_float_backend<50, allocate_stack> >   static_mpfr_float_50;
typedef number<mpfr_float_backend<100, allocate_stack> >  static_mpfr_float_100;

}} // namespaces
```

The `mpfr_float_backend` type is used in conjunction with `number`: It acts as a thin wrapper around the MPFR `mpfr_t` to provide an real-number type that is a drop-in replacement for the native C++ floating-point types, but with much greater precision.

Type `mpfr_float_backend` can be used at fixed precision by specifying a non-zero `Digits10` template parameter, or at variable precision by setting the template argument to zero. The typedefs mpfr_float_50, mpfr_float_100, mpfr_float_500, mpfr_float_1000 provide arithmetic types at 50, 100, 500 and 1000 decimal digits precision respectively. The typedef mpfr_float provides a variable precision type whose precision can be controlled via the `numbers` member functions.

In addition the second template parameter lets you choose between dynamic allocation (the default, and uses MPFR's normal allocation routines), or stack allocation (where all the memory required for the underlying data types is stored within `mpfr_float_backend`). The latter option can result in significantly faster code, at the expense of growing the size of `mpfr_float_backend`. It can only be used at fixed precision, and should only be used for lower digit counts. Note that we can not guarantee that using `allocate_stack` won't cause any calls to mpfr's allocation routines, as mpfr may call these inside it's own code. The following table gives an idea of the performance tradeoff's at 50 decimal digits precision[2]:

Type	Bessel function evaluation, relative times
`number<mpfr_float_backend<50, allocate_static>, et_on>`	1.0 (5.5s)
`number<mpfr_float_backend<50, allocate_static>, et_off>`	1.05 (5.8s)
`number<mpfr_float_backend<50, allocate_dynamic>, et_on>`	1.05 (5.8s)
`number<mpfr_float_backend<50, allocate_dynamic>, et_off>`	1.16 (6.4s)

> **Note**
>
> This type only provides `numeric_limits` support when the precision is fixed at compile time.

As well as the usual conversions from arithmetic and string types, instances of `number<mpfr_float_backend<N> >` are copy constructible and assignable from:

- The GMP native types `mpf_t`, `mpz_t`, `mpq_t`.

- The MPFR native type `mpfr_t`.

- The `number` wrappers around those types: `number<mpfr_float_backend<M> >`, `number<mpf_float<M> >`, `number<gmp_int>`, `number<gmp_rational>`.

It's also possible to access the underlying `mpfr_t` via the data() member function of `mpfr_float_backend`.

Things you should know when using this type:

- A default constructed `mpfr_float_backend` is set to a NaN (this is the default MPFR behavior).

- All operations use round to nearest.

- No changes are made to GMP or MPFR global settings, so this type can coexist with existing MPFR or GMP code.

- The code can equally use MPIR in place of GMP - indeed that is the preferred option on Win32.

- This backend supports rvalue-references and is move-aware, making instantiations of `number` on this backend move aware.

- Conversion from a string results in a `std::runtime_error` being thrown if the string can not be interpreted as a valid floating point number.

- Division by zero results in an infinity.

---

[2] Compiled with VC++10 and /Ox, with MPFR-3.0.0 and MPIR-2.3.0

MPFR example:

```cpp
#include <boost/multiprecision/mpfr.hpp>

using namespace boost::multiprecision;

// Operations at variable precision and no numeric_limits support:
mpfr_float a = 2;
mpfr_float::default_precision(1000);
std::cout << mpfr_float::default_precision() << std::endl;
std::cout << sqrt(a) << std::endl; // print root-2

// Operations at fixed precision and full numeric_limits support:
mpfr_float_100 b = 2;
std::cout << std::numeric_limits<mpfr_float_100>::digits << std::endl;
// We can use any C++ std lib function:
std::cout << log(b) << std::endl; // print log(2)
// We can also use any function from Boost.Math:
std::cout << boost::math::tgamma(b) << std::endl;
// These even work when the argument is an expression template:
std::cout << boost::math::tgamma(b * b) << std::endl;

// Access the underlying data:
mpfr_t r;
mpfr_init(r);
mpfr_set(r, b.backend().data(), GMP_RNDN);
```

float128

```cpp
#include <boost/multiprecision/float128.hpp>
```

```cpp
namespace boost{ namespace multiprecision{

class float128_backend;

typedef number<float128_backend, et_off>    float128;

}} // namespaces
```

The `float128` number type is a very thin wrapper around GCC's `__float128` or Intel's `_Quad` data types and provides an real-number type that is a drop-in replacement for the native C++ floating-point types, but with a 113 bit mantissa, and compatible with FORTRAN's 128-bit QUAD real.

All the usual standard library and `numeric_limits` support are available, performance should be equivalent to the underlying native types: for example the LINPACK benchmarks for GCC's `__float128` and `boost::multiprecision::float128` both achieved 5.6 MFLOPS[3].

As well as the usual conversions from arithmetic and string types, instances of `float128` are copy constructible and assignable from GCC's `__float128` and Intel's `_Quad` data types.

It's also possible to access the underlying `__float128` or `_Quad` type via the `data()` member function of `float128_backend`.

Things you should know when using this type:

• Default constructed `float128`s have the value zero.

• This backend supports rvalue-references and is move-aware, making instantiations of `number` on this backend move aware.

---

[3] On 64-bit Ubuntu 11.10, GCC-4.8.0, Intel Core 2 Duo T5800.

- It is not possible to round-trip objects of this type to and from a string and get back exactly the same value when compiled with Intel's C++ compiler: this is a current limitation of our code. Round tripping when using GCC is possible.

- Conversion from a string results in a `std::runtime_error` being thrown if the string can not be interpreted as a valid floating point number.

- Division by zero results in an infinity being produced.

- Type `float128` can be used as a literal type (constexpr support).

float128 example:

```cpp
#include <boost/multiprecision/float128.hpp>

using namespace boost::multiprecision;

// Operations at 128-bit precision and full numeric_limits support:
float128 b = 2;
// There are 113-bits of precision:
std::cout << std::numeric_limits<float128>::digits << std::endl;
// Or 34 decimal places:
std::cout << std::numeric_limits<float128>::digits10 << std::endl;
// We can use any C++ std lib function, lets print all the digits as well:
std::cout << std::setprecision(std::numeric_limits<float128>::max_digits10)
   << log(b) << std::endl; // print log(2) = 0.693147180559945309417232121458176575
// We can also use any function from Boost.Math:
std::cout << boost::math::tgamma(b) << std::endl;
// And since we have an extended exponent range we can generate some really large
// numbers here (4.02387260077093773543702433923004111e+2564):
std::cout << boost::math::tgamma(float128(1000)) << std::endl;
//
// We can declare constants using GCC or Intel's native types, and the Q suffix,
// these can be declared constexpr if required:

constexpr float128 pi = 3.1415926535897932384626433832795028841971693993751058Q;
```

Examples

Area of Circle

Generic numeric programming employs templates to use the same code for different floating-point types and functions. Consider the area of a circle a of radius r, given by

$$a = \pi * r^2$$

The area of a circle can be computed in generic programming using Boost.Math for the constant π as shown below:

```cpp
#include <boost/math/constants/constants.hpp>

template<typename T>
inline T area_of_a_circle(T r)
{
   using boost::math::constants::pi;
   return pi<T>() * r * r;
}
```

It is possible to use `area_of_a_circle()` with built-in floating-point types as well as floating-point types from Boost.Multiprecision. In particular, consider a system with 4-byte single-precision float, 8-byte double-precision double and also the `cpp_dec_float_50` data type from Boost.Multiprecision with 50 decimal digits of precision.

We can compute and print the approximate area of a circle with radius 123/100 for `float`, `double` and `cpp_dec_float_50` with the program below.

```cpp
#include <iostream>
#include <iomanip>
#include <boost/multiprecision/cpp_dec_float.hpp>

using boost::multiprecision::cpp_dec_float_50;

int main(int, char**)
{
   const float r_f(float(123) / 100);
   const float a_f = area_of_a_circle(r_f);

   const double r_d(double(123) / 100);
   const double a_d = area_of_a_circle(r_d);

   const cpp_dec_float_50 r_mp(cpp_dec_float_50(123) / 100);
   const cpp_dec_float_50 a_mp = area_of_a_circle(r_mp);

   // 4.75292
   std::cout
      << std::setprecision(std::numeric_limits<float>::digits10)
      << a_f
      << std::endl;

   // 4.752915525616
   std::cout
      << std::setprecision(std::numeric_limits<double>::digits10)
      << a_d
      << std::endl;

   // 4.7529155256159981904701331745635599135018975843146
   std::cout
      << std::setprecision(std::numeric_limits<cpp_dec_float_50>::digits10)
      << a_mp
      << std::endl;
}
```

In the next example we'll look at calling both standard library and Boost.Math functions from within generic code. We'll also show how to cope with template arguments which are expression-templates rather than number types.

Defining a Special Function.

In this example we'll show several implementations of the Jahnke and Emden Lambda function, each implementation a little more sophisticated than the last.

The Jahnke-Emden Lambda function is defined by the equation:

$$JahnkeEmden(v, z) = \Gamma(v+1) * J_v(z) / (z / 2)^v$$

If we were to implement this at double precision using Boost.Math's facilities for the Gamma and Bessel function calls it would look like this:

```cpp
double JEL1(double v, double z)
{
   return boost::math::tgamma(v + 1) * boost::math::cyl_bessel_j(v, z) / std::pow(z / 2, v);
}
```

Calling this function as:

```
std::cout << std::scientific << std::setprecision(std::numeric_limits<double>::digits10);
std::cout << JEL1(2.5, 0.5) << std::endl;
```

Yields the output:

```
9.822663964796047e-001
```

Now let's implement the function again, but this time using the multiprecision type `cpp_dec_float_50` as the argument type:

```
boost::multiprecision::cpp_dec_float_50
    JEL2(boost::multiprecision::cpp_dec_float_50 v, boost::multiprecision::cpp_dec_float_50 z)
{
    return boost::math::tgamma(v + 1) * boost::math::cyl_bessel_j(v, z) / boost::multipreci↵
sion::pow(z / 2, v);
}
```

The implementation is almost the same as before, but with one key difference - we can no longer call `std::pow`, instead we must call the version inside the `boost::multiprecision` namespace. In point of fact, we could have omitted the namespace prefix on the call to `pow` since the right overload would have been found via argument dependent lookup in any case.

Note also that the first argument to `pow` along with the argument to `tgamma` in the above code are actually expression templates. The `pow` and `tgamma` functions will handle these arguments just fine.

Here's an example of how the function may be called:

```
std::cout << std::scientific << std::setprecision(std::numeric_limits<cpp_dec_float_50>::digits10);
std::cout << JEL2(cpp_dec_float_50(2.5), cpp_dec_float_50(0.5)) << std::endl;
```

Which outputs:

```
9.8226639647960475701733500979688283399590 3762577173e-01
```

Now that we've seen some non-template examples, lets repeat the code again, but this time as a template that can be called either with a builtin type (`float`, `double` etc), or with a multiprecision type:

```
template <class Float>
Float JEL3(Float v, Float z)
{
    using std::pow;
    return boost::math::tgamma(v + 1) * boost::math::cyl_bessel_j(v, z) / pow(z / 2, v);
}
```

Once again the code is almost the same as before, but the call to `pow` has changed yet again. We need the call to resolve to either `std::pow` (when the argument is a builtin type), or to `boost::multiprecision::pow` (when the argument is a multiprecision type). We do that by making the call unqualified so that versions of `pow` defined in the same namespace as type `Float` are found via argument dependent lookup, while the `using std::pow` directive makes the standard library versions visible for builtin floating point types.

Let's call the function with both `double` and multiprecision arguments:

```
std::cout << std::scientific << std::setprecision(std::numeric_limits<double>::digits10);
std::cout << JEL3(2.5, 0.5) << std::endl;
std::cout << std::scientific << std::setprecision(std::numeric_limits<cpp_dec_float_50>::digits10);
std::cout << JEL3(cpp_dec_float_50(2.5), cpp_dec_float_50(0.5)) << std::endl;
```

Which outputs:

```
9.822663964796047e-001
9.8226639647960475701733500979688283399590376257717 3e-01
```

Unfortunately there is a problem with this version: if we were to call it like this:

```
boost::multiprecision::cpp_dec_float_50 v(2), z(0.5);
JEL3(v + 0.5, z);
```

Then we would get a long and inscrutable error message from the compiler: the problem here is that the first argument to JEL3 is not a number type, but an expression template. We could obviously add a typecast to fix the issue:

```
JEL(cpp_dec_float_50(v + 0.5), z);
```

However, if we want the function JEL to be truly reusable, then a better solution might be preferred. To achieve this we can borrow some code from Boost.Math which calculates the return type of mixed-argument functions, here's how the new code looks now:

```
template <class Float1, class Float2>
typename boost::math::tools::promote_args<Float1, Float2>::type
   JEL4(Float1 v, Float2 z)
{
   using std::pow;
   return boost::math::tgamma(v + 1) * boost::math::cyl_bessel_j(v, z) / pow(z / 2, v);
}
```

As you can see the two arguments to the function are now separate template types, and the return type is computed using the `promote_args` metafunction from Boost.Math.

Now we can call:

```
std::cout << std::scientific << std::setprecision(std::numeric_limits<cpp_dec_float_100>::di↵
gits10);
std::cout << JEL4(cpp_dec_float_100(2) + 0.5, cpp_dec_float_100(0.5)) << std::endl;
```

And get 100 digits of output:

```
9.822663964796047570173350097968828339959037625771730906941041382216508224815363845414700423684891 7775e-
01
```

As a bonus, we can now call the function not just with expression templates, but with other mixed types as well: for example `float` and `double` or `int` and `double`, and the correct return type will be computed in each case.

Note that while in this case we didn't have to change the body of the function, in the general case any function like this which creates local variables internally would have to use `promote_args` to work out what type those variables should be, for example:

```
template <class Float1, class Float2>
typename boost::math::tools::promote_args<Float1, Float2>::type
   JEL5(Float1 v, Float2 z)
{
   using std::pow;
   typedef typename boost::math::tools::promote_args<Float1, Float2>::type variable_type;
   variable_type t = pow(z / 2, v);
   return boost::math::tgamma(v + 1) * boost::math::cyl_bessel_j(v, z) / t;
}
```

Calculating a Derivative

In this example we'll add even more power to generic numeric programming using not only different floating-point types but also function objects as template parameters. Consider some well-known central difference rules for numerically computing the first derivative of a function $f'(x)$ with $x \in \Re$:

$$(1) \quad \begin{aligned} f'(x) &\approx m_1 + O(dx^2) \\ f'(x) &\approx \tfrac{4}{3}m_1 - \tfrac{1}{3}m_2 + O(dx^4) \\ f'(x) &\approx \tfrac{3}{2}m_1 - \tfrac{3}{5}m_2 + \tfrac{1}{10}m_3 + O(dx^6) \end{aligned}$$

Where the difference terms m_n are given by:

$$(2) \quad \begin{aligned} m_1 &= \frac{f(x+dx)-f(x-dx)}{2dx} \\ m_2 &= \frac{f(x+2dx)-f(x-2dx)}{4dx} \\ m_3 &= \frac{f(x+3dx)-f(x-3dx)}{6dx} \end{aligned}$$

and dx is the step-size of the derivative.

The third formula in Equation 1 is a three-point central difference rule. It calculates the first derivative of $f'(x)$ to $O(dx^6)$, where dx is the given step-size. For example, if the step-size is 0.01 this derivative calculation has about 6 decimal digits of precision - just about right for the 7 decimal digits of single-precision float. Let's make a generic template subroutine using this three-point central difference rule. In particular:

```cpp
template<typename value_type, typename function_type>
    value_type derivative(const value_type x, const value_type dx, function_type func)
{
    // Compute d/dx[func(*first)] using a three-point
    // central difference rule of O(dx^6).

    const value_type dx1 = dx;
    const value_type dx2 = dx1 * 2;
    const value_type dx3 = dx1 * 3;

    const value_type m1 = (func(x + dx1) - func(x - dx1)) / 2;
    const value_type m2 = (func(x + dx2) - func(x - dx2)) / 4;
    const value_type m3 = (func(x + dx3) - func(x - dx3)) / 6;

    const value_type fifteen_m1 = 15 * m1;
    const value_type six_m2     =  6 * m2;
    const value_type ten_dx1    = 10 * dx1;

    return ((fifteen_m1 - six_m2) + m3) / ten_dx1;
}
```

The `derivative()` template function can be used to compute the first derivative of any function to $O(dx^6)$. For example, consider the first derivative of *sin(x)* evaluated at $x = \pi/3$. In other words,

$$(3) \quad \frac{d}{dx}\sin x\Big|_{x=\frac{\pi}{3}} = \cos\frac{\pi}{3} = \frac{1}{2}$$

The code below computes the derivative in Equation 3 for float, double and boost's multiple-precision type cpp_dec_float_50.

```cpp
#include <iostream>
#include <iomanip>
#include <boost/multiprecision/cpp_dec_float.hpp>
#include <boost/math/constants/constants.hpp>


int main(int, char**)
{
   using boost::math::constants::pi;
   using boost::multiprecision::cpp_dec_float_50;
   //
   // We'll pass a function pointer for the function object passed to derivative,
   // the typecast is needed to select the correct overload of std::sin:
   //
   const float d_f = derivative(
      pi<float>() / 3,
      0.01F,
      static_cast<float(*)(float)>(std::sin)
   );

   const double d_d = derivative(
      pi<double>() / 3,
      0.001,
      static_cast<double(*)(double)>(std::sin)
      );
   //
   // In the cpp_dec_float_50 case, the sin function is multiply overloaded
   // to handle expression templates etc.  As a result it's hard to take its
   // address without knowing about its implementation details.  We'll use a
   // C++11 lambda expression to capture the call.
   // We also need a typecast on the first argument so we don't accidentally pass
   // an expression template to a template function:
   //
   const cpp_dec_float_50 d_mp = derivative(
      cpp_dec_float_50(pi<cpp_dec_float_50>() / 3),
      cpp_dec_float_50(1.0E-9),
      [](const cpp_dec_float_50& x) -> cpp_dec_float_50
      {
         return sin(x);
      }
      );

   // 5.000029e-001
   std::cout
      << std::setprecision(std::numeric_limits<float>::digits10)
      << d_f
      << std::endl;

   // 4.99999999998876e-001
   std::cout
      << std::setprecision(std::numeric_limits<double>::digits10)
      << d_d
      << std::endl;

   // 4.9999999999999999999999999999999999999999999999999e-01
   std::cout
      << std::setprecision(std::numeric_limits<cpp_dec_float_50>::digits10)
      << d_mp
      << std::endl;
}
```

The expected value of the derivative is 0.5. This central difference rule in this example is ill-conditioned, meaning it suffers from slight loss of precision. With that in mind, the results agree with the expected value of 0.5.

We can take this a step further and use our derivative function to compute a partial derivative. For example if we take the incomplete gamma function *P(a, z)*, and take the derivative with respect to *z* at *(2,2)* then we can calculate the result as shown below, for good measure we'll compare with the "correct" result obtained from a call to *gamma_p_derivative*, the results agree to approximately 44 digits:

```cpp
cpp_dec_float_50 gd = derivative(
   cpp_dec_float_50(2),
   cpp_dec_float_50(1.0E-9),
   [](const cpp_dec_float_50& x) ->cpp_dec_float_50
   {
      return boost::math::gamma_p(2, x);
   }
);
// 2.7067056647322538378799898994496880681526309181915le-01
std::cout
   << std::setprecision(std::numeric_limits<cpp_dec_float_50>::digits10)
   << gd
   << std::endl;
// 2.7067056647322538378799898994496880681525319014312e-01
std::cout << boost::math::gamma_p_derivat↵
ive(cpp_dec_float_50(2), cpp_dec_float_50(2)) << std::endl;
```

Calculating an Integral

Similar to the generic derivative example, we can calculate integrals in a similar manner:

```
template<typename value_type, typename function_type>
inline value_type integral(const value_type a,
                           const value_type b,
                           const value_type tol,
                           function_type func)
{
   unsigned n = 1U;

   value_type h = (b - a);
   value_type I = (func(a) + func(b)) * (h / 2);

   for(unsigned k = 0U; k < 8U; k++)
   {
      h /= 2;

      value_type sum(0);
      for(unsigned j = 1U; j <= n; j++)
      {
         sum += func(a + (value_type((j * 2) - 1) * h));
      }

      const value_type I0 = I;
      I = (I / 2) + (h * sum);

      const value_type ratio     = I0 / I;
      const value_type delta     = ratio - 1;
      const value_type delta_abs = ((delta < 0) ? -delta : delta);

      if((k > 1U) && (delta_abs < tol))
      {
         break;
      }

      n *= 2U;
   }

   return I;
}
```

The following sample program shows how the function can be called, we begin by defining a function object, which when integrated should yield the Bessel J function:

```
template<typename value_type>
class cyl_bessel_j_integral_rep
{
public:
   cyl_bessel_j_integral_rep(const unsigned N,
      const value_type& X) : n(N), x(X) { }

   value_type operator()(const value_type& t) const
   {
      // pi * Jn(x) = Int_0^pi [cos(x * sin(t) - n*t) dt]
      return cos(x * sin(t) - (n * t));
   }

private:
   const unsigned n;
   const value_type x;
};
```

```
   /* The function can now be called as follows: */
int main(int, char**)
{
   using boost::math::constants::pi;
   typedef boost::multiprecision::cpp_dec_float_50 mp_type;

   const float j2_f =
      integral(0.0F,
      pi<float>(),
      0.01F,
      cyl_bessel_j_integral_rep<float>(2U, 1.23F)) / pi<float>();

   const double j2_d =
      integral(0.0,
      pi<double>(),
      0.0001,
      cyl_bessel_j_integral_rep<double>(2U, 1.23)) / pi<double>();

   const mp_type j2_mp =
      integral(mp_type(0),
      pi<mp_type>(),
      mp_type(1.0E-20),
      cyl_bessel_j_integral_rep<mp_type>(2U, mp_type(123) / 100)) / pi<mp_type>();

   // 0.166369
   std::cout
      << std::setprecision(std::numeric_limits<float>::digits10)
      << j2_f
      << std::endl;

   // 0.166369383786814
   std::cout
      << std::setprecision(std::numeric_limits<double>::digits10)
      << j2_d
      << std::endl;

   // 0.16636938378681407351267852431513159437103348245333
   std::cout
      << std::setprecision(std::numeric_limits<mp_type>::digits10)
      << j2_mp
      << std::endl;

   //
   // Print true value for comparison:
   // 0.16636938378681407351267852431513159437103482453329
   std::cout << boost::math::cyl_bessel_j(2, mp_type(123) / 100) << std::endl;
}
```

Polynomial Evaluation

In this example we'll look at polynomial evaluation, this is not only an important use case, but it's one that `number` performs particularly well at because the expression templates *completely eliminate all temporaries* from a Horner polynomial evaluation scheme.

The following code evaluates `sin(x)` as a polynomial, accurate to at least 64 decimal places:

```cpp
using boost::multiprecision::cpp_dec_float;
typedef boost::multiprecision::number<cpp_dec_float<64> > mp_type;

mp_type mysin(const mp_type& x)
{
  // Approximation of sin(x * pi/2) for -1 <= x <= 1, using an order 63 polynomial.
  static const std::array<mp_type, 32U> coefs =
  {{

 mp_type("+1.5707963267948966192313216916397514420985846996875529104874722961539082031431044993140174126711"), //"),

 mp_type("-0.64596409750624625365575656389794573337969351178927307696134454382929989411386887578263960484"), //
^3

 mp_type("+0.07969262624616704512050554949047802252091164235106119545663865720995702920146198554317279"), //
^5

 mp_type("-0.0046817541353186881006854639339534378594950280185010575749538605102665157913157426229824"), //
^7

 mp_type("+0.00016044118478735982187266087016347332970280754062061156858775174056686380286868007443"), //
^9
    mp_type("-3.598843235212085340458540018208389404888495232432127661083907575106196374913134E-
6"), // ^11
    mp_type("+5.692172921967926811775255303592184372902829756054598109818158853197797542565E-
8"), // ^13
    mp_type("-6.688035109811467232478226335783138689956270985704278659373558497256423498E-
10"), // ^15
    mp_type("+6.066935731106195667101445665327140070166203261129845646380005577490472E-
12"), // ^17
    mp_type("-4.377065467313742277184271313776319094862897030084226361576452003432E-14"), // ^19
    mp_type("+2.571422892860473866153865950420487369167895373255729246889168337E-16"), // ^21
    mp_type("-1.253899540535457665340073300390626396596970180355253776711660E-18"), // ^23
    mp_type("+5.15645517658028233395375998562329055050964428219501277474E-21"), // ^25
    mp_type("-1.812399312848887477410034071087545686586497030654642705E-23"), // ^27
    mp_type("+5.5072857865223858357085513920522536675023562254864E-26"), // ^29
    mp_type("-1.461148710664467988723468673933026649943084902958E-28"), // ^31
    mp_type("+3.41405297003316172502972039913417222912445427E-31"), // ^33
    mp_type("-7.07885550810745570069916712806856538290251E-34"), // ^35
    mp_type("+1.31128947968267628970845439024155655665E-36"), // ^37
    mp_type("-2.18318293181145698535113946654065918E-39"), // ^39
    mp_type("+3.28462680978498856345937578502923E-42"), // ^41
    mp_type("-4.48753699028101089490067137298E-45"), // ^43
    mp_type("+5.59219884208696457859353716E-48"), // ^45
    mp_type("-6.38214503973500471720565E-51"), // ^47
    mp_type("+6.69528558381794452556E-54"), // ^49
    mp_type("-6.47841373182350206E-57"), // ^51
    mp_type("+5.800016389666445E-60"), // ^53
    mp_type("-4.818507347289E-63"), // ^55
    mp_type("+3.724683686E-66"), // ^57
    mp_type("-2.6856479E-69"), // ^59
    mp_type("+1.81046E-72"), // ^61
    mp_type("-1.133E-75"), // ^63
  }};

  const mp_type v = x * 2 / boost::math::constants::pi<mp_type>();
  const mp_type x2 = (v * v);
  //
  // Polynomial evaluation follows, if mp_type allocates memory then
  // just one such allocation occurs - to initialize the variable "sum" -
  // and no temporaries are created at all.
  //
  const mp_type sum = ((((((((((((((((((((((((((((((((       + coefs[31U]
```

```
                                                        * x2 + coefs[30U])
                                                        * x2 + coefs[29U])
                                                        * x2 + coefs[28U])
                                                        * x2 + coefs[27U])
                                                        * x2 + coefs[26U])
                                                        * x2 + coefs[25U])
                                                        * x2 + coefs[24U])
                                                        * x2 + coefs[23U])
                                                        * x2 + coefs[22U])
                                                        * x2 + coefs[21U])
                                                        * x2 + coefs[20U])
                                                        * x2 + coefs[19U])
                                                        * x2 + coefs[18U])
                                                        * x2 + coefs[17U])
                                                        * x2 + coefs[16U])
                                                        * x2 + coefs[15U])
                                                        * x2 + coefs[14U])
                                                        * x2 + coefs[13U])
                                                        * x2 + coefs[12U])
                                                        * x2 + coefs[11U])
                                                        * x2 + coefs[10U])
                                                        * x2 + coefs[9U])
                                                        * x2 + coefs[8U])
                                                        * x2 + coefs[7U])
                                                        * x2 + coefs[6U])
                                                        * x2 + coefs[5U])
                                                        * x2 + coefs[4U])
                                                        * x2 + coefs[3U])
                                                        * x2 + coefs[2U])
                                                        * x2 + coefs[1U])
                                                        * x2 + coefs[0U])
                                                        * v;

  return sum;
}
```

Calling the function like so:

```
mp_type pid4 = boost::math::constants::pi<mp_type>() / 4;
std::cout << std::setprecision(std::numeric_limits< ::mp_type>::digits10) << std::scientific;
std::cout << mysin(pid4) << std::endl;
```

Yields the expected output:

```
7.0710678118654752440084436210484903928483593768847403658833986900e-01
```

Interval Number Types

There is one currently only one interval number type supported - MPFI.

mpfi_float

```
#include <boost/multiprecision/mpfi.hpp>
```

```
namespace boost{ namespace multiprecision{

template <unsigned Digits10>
class mpfi_float_backend;

typedef number<mpfi_float_backend<50> >    mpfi_float_50;
typedef number<mpfi_float_backend<100> >   mpfifloat_100;
typedef number<mpfi_float_backend<500> >   mpfifloat_500;
typedef number<mpfi_float_backend<1000> >  mpfi_float_1000;
typedef number<mpfi_float_backend<0> >     mpfi_float;

}} // namespaces
```

The `mpfi_float_backend` type is used in conjunction with `number`: It acts as a thin wrapper around the MPFI `mpfi_t` to provide an real-number type that is a drop-in replacement for the native C++ floating-point types, but with much greater precision and implementing interval arithmetic.

Type `mpfi_float_backend` can be used at fixed precision by specifying a non-zero `Digits10` template parameter, or at variable precision by setting the template argument to zero. The typedefs mpfi_float_50, mpfi_float_100, mpfi_float_500, mpfi_float_1000 provide arithmetic types at 50, 100, 500 and 1000 decimal digits precision respectively. The typedef mpfi_float provides a variable precision type whose precision can be controlled via the `numbers` member functions.

> **Note**
>
> This type only provides `numeric_limits` support when the precision is fixed at compile time.

As well as the usual conversions from arithmetic and string types, instances of `number<mpfi_float_backend<N> >` are copy constructible and assignable from:

- The MPFI native type `mpfi_t`.

- The `number` wrappers around MPFI or MPFR: `number<mpfi_float_backend<M> >` and `number<mpfr_float<M> >`.

- There is a two argument constructor taking two `number<mpfr_float<M> >` arguments specifying the interval.

It's also possible to access the underlying `mpfi_t` via the data() member function of `mpfi_float_backend`.

Things you should know when using this type:

- A default constructed `mpfi_float_backend` is set to a NaN (this is the default MPFI behavior).

- No changes are made to GMP or MPFR global settings, so this type can coexist with existing MPFR or GMP code.

- The code can equally use MPIR in place of GMP - indeed that is the preferred option on Win32.

- This backend supports rvalue-references and is move-aware, making instantiations of `number` on this backend move aware.

- Conversion from a string results in a `std::runtime_error` being thrown if the string can not be interpreted as a valid floating point number.

- Division by zero results in an infinity.

There are some additional non member functions for working on intervals:

```
template <unsigned Digits10, expression_template_option ExpressionTemplates>
number<mpfr_float_backend<Digits10>, ExpressionTemplates> lower(const number<mpfi_float_backend<Di↵
gits10>, ExpressionTemplates>& val);
```

Returns the lower end of the interval.

```
template <unsigned Digits10, expression_template_option ExpressionTemplates>
number<mpfr_float_backend<Digits10>, ExpressionTemplates> upper(const number<mpfi_float_backend<Di↵
gits10>, ExpressionTemplates>& val);
```

Returns the upper end of the interval.

```
template <unsigned Digits10, expression_template_option ExpressionTemplates>
number<mpfr_float_backend<Digits10>, ExpressionTemplates> median(const number<mpfi_float_backend<Di↵
gits10>, ExpressionTemplates>& val);
```

Returns the mid point of the interval.

```
template <unsigned Digits10, expression_template_option ExpressionTemplates>
number<mpfr_float_backend<Digits10>, ExpressionTemplates> width(const number<mpfi_float_backend<Di↵
gits10>, ExpressionTemplates>& val);
```

Returns the absolute width of the interval.

```
template <unsigned Digits10, expression_template_option ExpressionTemplates>
number<mpfi_float_backend<Digits10>, ExpressionTemplates> intersect(
 const number<mpfi_float_backend<Digits10>, ExpressionTemplates>& a,
 const number<mpfi_float_backend<Digits10>, ExpressionTemplates>&  b);
```

Returns the interval which is the intersection of the *a* and *b*. Returns an unspecified empty interval if there is no such intersection.

```
template <unsigned Digits10, expression_template_option ExpressionTemplates>
number<mpfi_float_backend<Digits10>, ExpressionTemplates> hull(
 const number<mpfi_float_backend<Digits10>, ExpressionTemplates>& a,
 const number<mpfi_float_backend<Digits10>, ExpressionTemplates>&  b);
```

Returns the interval which is the union of *a* and *b*.

```
template <unsigned Digits10, expression_template_option ExpressionTemplates>
bool overlap(const number<mpfi_float_backend<Digits10>, ExpressionTemplates>& a,
             const number<mpfi_float_backend<Digits10>, ExpressionTemplates>&  b);
```

Returns `true` only if the intervals *a* and *b* overlap.

```
template <unsigned Digits10, expression_template_option ExpressionTemplates1, expression_tem↵
plate_option ExpressionTemplates2>
bool in(const number<mpfr_float_backend<Digits10>, ExpressionTemplates1>& a,
        const number<mpfi_float_backend<Digits10>, ExpressionTemplates2>&  b);
```

Returns `true` only if point *a* is contained within the interval *b*.

```
template <unsigned Digits10, expression_template_option ExpressionTemplates>
bool zero_in(const number<mpfi_float_backend<Digits10>, ExpressionTemplates>& a);
```

Returns `true` only if the interval *a* contains the value zero.

```
template <unsigned Digits10, expression_template_option ExpressionTemplates>
bool subset(const number<mpfi_float_backend<Digits10>, ExpressionTemplates>& a,
            const number<mpfi_float_backend<Digits10>, ExpressionTemplates>& b);
```

Returns `true` only if *a* is a subset of *b*.

```
template <unsigned Digits10, expression_template_option ExpressionTemplates>
bool proper_subset(const number<mpfi_float_backend<Digits10>, ExpressionTemplates>& a,
                   const number<mpfi_float_backend<Digits10>, ExpressionTemplates>& b);
```

Returns `true` only if *a* is a proper subset of *b*.

```
template <unsigned Digits10, expression_template_option ExpressionTemplates>
bool empty(const number<mpfi_float_backend<Digits10>, ExpressionTemplates>& a);
```

Returns `true` only if *a* is an empty interval, equivalent to `upper(a) < lower(a)`.

```
template <unsigned Digits10, expression_template_option ExpressionTemplates>
bool singleton(const number<mpfi_float_backend<Digits10>, ExpressionTemplates>& a);
```

Returns `true` if `lower(a) == upper(a)`.

MPFI example:

```cpp
#include <boost/multiprecision/mpfi.hpp>

using namespace boost::multiprecision;

// Operations at variable precision and no numeric_limits support:
mpfi_float a = 2;
mpfi_float::default_precision(1000);
std::cout << mpfi_float::default_precision() << std::endl;
std::cout << sqrt(a) << std::endl; // print root-2

// Operations at fixed precision and full numeric_limits support:
mpfi_float_100 b = 2;
std::cout << std::numeric_limits<mpfi_float_100>::digits << std::endl;
// We can use any C++ std lib function:
std::cout << log(b) << std::endl; // print log(2)

// Access the underlying data:
mpfi_t r;
mpfi_init(r);
mpfi_set(r, b.backend().data());

// Construct some explicit intervals and perform set operations:
mpfi_float_50 i1(1, 2), i2(1.5, 2.5);
std::cout << intersect(i1, i2) << std::endl;
std::cout << hull(i1, i2) << std::endl;
std::cout << overlap(i1, i2) << std::endl;
std::cout << subset(i1, i2) << std::endl;
```

Rational Number Types

The following back-ends provide rational number arithmetic:

Backend Type	Header	Radix	Dependencies	Pros	Cons
`cpp_rational`	boost/multiprecision/cpp_int.hpp	2	None	An all C++ Boost-licensed implementation.	Slower than GMP.
`gmp_rational`	boost/multiprecision/gmp.hpp	2	GMP	Very fast and efficient back-end.	Dependency on GNU licensed GMP library.
`tommath_rational`	boost/multiprecision/tommath.hpp	2	libtommath	All C/C++ implementation that's Boost Software Licence compatible.	Slower than GMP.
`rational_adaptor`	boost/multiprecision/rational_adaptor.hpp	N/A	none	All C++ adaptor that allows any integer back-end type to be used as a rational type.	Requires an underlying integer back-end type.
`boost::rational`	boost/rational.hpp	N/A	None	A C++ rational number type that can used with any `number` integer type.	The expression templates used by `number` end up being "hidden" inside `boost::rational`: performance may well suffer as a result.

cpp_rational

```
#include <boost/multiprecision/cpp_int.hpp>
```

```
namespace boost{ namespace multiprecision{

typedef rational_adaptor<cpp_int_backend<> >    cpp_rational_backend;

typedef number<cpp_rational_backend>         cpp_rational;

}} // namespaces
```

The `cpp_rational_backend` type is used via the typedef `boost::multiprecision::cpp_rational`. It provides a rational number type that is a drop-in replacement for the native C++ number types, but with unlimited precision.

As well as the usual conversions from arithmetic and string types, instances of `cpp_rational` are copy constructible and assignable from type `cpp_int`.

There is also a two argument constructor that accepts a numerator and denominator: both of type `cpp_int`.

There are also non-member functions:

```
cpp_int numerator(const cpp_rational&);
cpp_int denominator(const cpp_rational&);
```

which return the numerator and denominator of the number.

Things you should know when using this type:

- Default constructed `cpp_rationals` have the value zero.

- Division by zero results in a `std::overflow_error` being thrown.

- Conversion from a string results in a `std::runtime_error` being thrown if the string can not be interpreted as a valid rational number.

Example:

```cpp
#include <boost/multiprecision/cpp_int.hpp>

using namespace boost::multiprecision;

cpp_rational v = 1;

// Do some arithmetic:
for(unsigned i = 1; i <= 1000; ++i)
   v *= i;
v /= 10;

std::cout << v << std::endl; // prints 1000! / 10
std::cout << numerator(v) << std::endl;
std::cout << denominator(v) << std::endl;

cpp_rational w(2, 3);  // component wise constructor
std::cout << w << std::endl; // prints 2/3
```

gmp_rational

```cpp
#include <boost/multiprecision/gmp.hpp>
```

```cpp
namespace boost{ namespace multiprecision{

class gmp_rational;

typedef number<gmp_rational >       mpq_rational;

}} // namespaces
```

The `gmp_rational` back-end is used via the typedef `boost::multiprecision::mpq_rational`. It acts as a thin wrapper around the GMP `mpq_t` to provide a rational number type that is a drop-in replacement for the native C++ number types, but with unlimited precision.

As well as the usual conversions from arithmetic and string types, instances of `number<gmp_rational>` are copy constructible and assignable from:

- The GMP native types: `mpz_t`, `mpq_t`.

- `number<gmp_int>`.

There is also a two-argument constructor that accepts a numerator and denominator (both of type `number<gmp_int>`).

There are also non-member functions:

```cpp
mpz_int numerator(const mpq_rational&);
mpz_int denominator(const mpq_rational&);
```

which return the numerator and denominator of the number.

It's also possible to access the underlying `mpq_t` via the `data()` member function of `mpq_rational`.

Things you should know when using this type:

- Default constructed `mpq_rationals` have the value zero (this is the GMP default behavior).

- Division by zero results in a `std::overflow_error` being thrown.

- Conversion from a string results in a `std::runtime_error` being thrown if the string can not be interpreted as a valid rational number.

- No changes are made to the GMP library's global settings, so this type can coexist with existing GMP code.

- The code can equally be used with MPIR as the underlying library - indeed that is the preferred option on Win32.

Example:

```cpp
#include <boost/multiprecision/gmp.hpp>

using namespace boost::multiprecision;

mpq_rational v = 1;

// Do some arithmetic:
for(unsigned i = 1; i <= 1000; ++i)
   v *= i;
v /= 10;

std::cout << v << std::endl; // prints 1000! / 10
std::cout << numerator(v) << std::endl;
std::cout << denominator(v) << std::endl;

mpq_rational w(2, 3);  // component wise constructor
std::cout << w << std::endl; // prints 2/3

// Access the underlying data:
mpq_t q;
mpq_init(q);
mpq_set(q, v.backend().data());
```

tommath_rational

```cpp
#include <boost/multiprecision/tommath.hpp>
```

```cpp
namespace boost{ namespace multiprecision{

typedef rational_adpater<tommath_int>        tommath_rational;
typedef number<tommath_rational >       tom_rational;

}} // namespaces
```

The `tommath_rational` back-end is used via the typedef `boost::multiprecision::tom_rational`. It acts as a thin wrapper around `boost::rational<tom_int>` to provide a rational number type that is a drop-in replacement for the native C++ number types, but with unlimited precision.

The advantage of using this type rather than `boost::rational<tom_int>` directly, is that it is expression-template enabled, greatly reducing the number of temporaries created in complex expressions.

There are also non-member functions:

```
tom_int numerator(const tom_rational&);
tom_int denominator(const tom_rational&);
```

which return the numerator and denominator of the number.

Things you should know when using this type:

• Default constructed `tom_rationals` have the value zero (this the inherited Boost.Rational behavior).

• Division by zero results in a `std::overflow_error` being thrown.

• Conversion from a string results in a `std::runtime_error` being thrown if the string can not be interpreted as a valid rational number.

• No changes are made to libtommath's global state, so this type can safely coexist with other libtommath code.

• Performance of this type has been found to be pretty poor - this need further investigation - but it appears that Boost.Rational needs some improvement in this area.

Example:

```cpp
#include <boost/multiprecision/tommath.hpp>

using namespace boost::multiprecision;

tom_rational v = 1;

// Do some arithmetic:
for(unsigned i = 1; i <= 1000; ++i)
   v *= i;
v /= 10;

std::cout << v << std::endl; // prints 1000! / 10
std::cout << numerator(v) << std::endl;
std::cout << denominator(v) << std::endl;

tom_rational w(2, 3); // Component wise constructor
std::cout << w << std::endl; // prints 2/3
```

Use With Boost.Rational

All of the integer types in this library can be used as template arguments to `boost::rational<IntType>`.

Note that using the library in this way largely negates the effect of the expression templates in `number`.

rational_adaptor

```cpp
namespace boost{ namespace multiprecision{

template <class IntBackend>
class rational_adpater;

}}
```

The class template `rational_adaptor` is a back-end for `number` which converts any existing integer back-end into a rational-number back-end.

So for example, given an integer back-end type `MyIntegerBackend`, the use would be something like:

```
typedef number<MyIntegerBackend>                    MyInt;
typedef number<rational_adaptor<MyIntegerBackend> > MyRational;

MyRational r = 2;
r /= 3;
MyInt i = numerator(r);
assert(i == 2);
```

Miscellaneous Number Types.

Backend types listed in this section are predominantly designed to aid debugging.

logged_adaptor

```
#include <boost/multiprecision/logged_adaptor.hpp>
```

```
namespace boost{ namespace multiprecision{

template <class Backend>
void log_postfix_event(const Backend& result, const char* event_description);
template <class Backend, class T>
void log_postfix_event(const Backend& result1, const T& result2, const char* event_description);

template <class Backend>
void log_prefix_event(const Backend& arg1, const char* event_description);
template <class Backend, class T>
void log_prefix_event(const Backend& arg1, const T& arg2, const char* event_description);
template <class Backend, class T, class U>
void log_prefix_event(const Backend& arg1, const T& arg2, const U& arg3, const char* event_de↵
scription);
template <class Backend, class T, class U, class V>
void log_pre↵
fix_event(const Backend& arg1, const T& arg2, const U& arg3, const V& arg4, const char* event_de↵
scription);

template <Backend>
class logged_adaptor;

}} // namespaces
```

The `logged_adaptor` type is used in conjunction with `number` and some other backend type: it acts as a thin wrapper around some other backend to class `number` and logs all the events that take place on that object. Before any number operation takes place, it calls `log_prefix_event` with the arguments to the operation (up to 4), plus a string describing the operation. Then after the operation it calls `log_postfix_event` with the result of the operation, plus a string describing the operation. Optionally, `log_postfix_event` takes a second result argument: this occurs when the result of the operation is not a `number`, for example when `fpclassify` is called, `log_postfix_event` will be called with `result1` being the argument to the function, and `result2` being the integer result of `fpclassify`.

The default versions of `log_prefix_event` and `log_postfix_event` do nothing, it is therefore up to the user to overload these for the particular backend being observed.

This type provides `numeric_limits` support whenever the template argument Backend does so.

This type is particularly useful when combined with an interval number type - in this case we can use `log_postfix_event` to monitor the error accumulated after each operation. We could either set some kind of trap whenever the accumulated error exceeds some threshold, or simply print out diagnostic information. Using this technique we can quickly locate the cause of numerical instability in a particular routine. The following example demonstrates this technique in a trivial algorithm that deliberately introduces cancellation error:

```cpp
#include <boost/multiprecision/mpfi.hpp>
#include <boost/multiprecision/logged_adaptor.hpp>
#include <iostream>
#include <iomanip>
//
// Begin by overloading log_postfix_event so we can capture each arithmetic event as it happens:
//
namespace boost{ namespace multiprecision{

template <unsigned D>
inline void log_postfix_event(const mpfi_float_backend<D>& val, const char* event_description)
{
   // Print out the (relative) diameter of the interval:
   using namespace boost::multiprecision;
   number<mpfr_float_backend<D> > diam;
   mpfi_diam(diam.backend().data(), val.data());
   std::cout << "Diameter was " << diam << " after operation: " << event_description << std::endl;
}
template <unsigned D, class T>
inline void log_postfix_event(const mpfi_float_backend<D>&, const T&, const char* event_descrip‍
tion)
{
   // This version is never called in this example.
}

}}


int main()
{
   using namespace boost::multiprecision;
   typedef number<logged_adaptor<mpfi_float_backend<17> > > logged_type;
   //
   // Test case deliberately introduces cancellation error, relative size of interval
   // gradually gets larger after each operation:
   //
   logged_type a = 1;
   a /= 10;

   for(unsigned i = 0; i < 13; ++i)
   {
      logged_type b = a * 9;
      b /= 10;
      a -= b;
   }
   std::cout << "Final value was: " << a << std::endl;
   return 0;
}
```

When we examine program output we can clearly see that the diameter of the interval increases after each subtraction:

```
Diameter was nan after operation: Default construct
Diameter was 0 after operation: Assignment from arithmetic type
Diameter was 4.33681e-18 after operation: /=
Diameter was nan after operation: Default construct
Diameter was 7.70988e-18 after operation: *
Diameter was 9.63735e-18 after operation: /=
Diameter was 1.30104e-16 after operation: -=
Diameter was nan after operation: Default construct
Diameter was 1.30104e-16 after operation: *
Diameter was 1.38537e-16 after operation: /=
Diameter was 2.54788e-15 after operation: -=
Diameter was nan after operation: Default construct
Diameter was 2.54788e-15 after operation: *
Diameter was 2.54863e-15 after operation: /=
Diameter was 4.84164e-14 after operation: -=
Diameter was nan after operation: Default construct
Diameter was 4.84164e-14 after operation: *
Diameter was 4.84221e-14 after operation: /=
Diameter was 9.19962e-13 after operation: -=
Diameter was nan after operation: Default construct
Diameter was 9.19962e-13 after operation: *
Diameter was 9.19966e-13 after operation: /=
Diameter was 1.74793e-11 after operation: -=
Diameter was nan after operation: Default construct
Diameter was 1.74793e-11 after operation: *
Diameter was 1.74793e-11 after operation: /=
Diameter was 3.32107e-10 after operation: -=
Diameter was nan after operation: Default construct
Diameter was 3.32107e-10 after operation: *
Diameter was 3.32107e-10 after operation: /=
Diameter was 6.31003e-09 after operation: -=
Diameter was nan after operation: Default construct
Diameter was 6.31003e-09 after operation: *
Diameter was 6.31003e-09 after operation: /=
Diameter was 1.19891e-07 after operation: -=
Diameter was nan after operation: Default construct
Diameter was 1.19891e-07 after operation: *
Diameter was 1.19891e-07 after operation: /=
Diameter was 2.27792e-06 after operation: -=
Diameter was nan after operation: Default construct
Diameter was 2.27792e-06 after operation: *
Diameter was 2.27792e-06 after operation: /=
Diameter was 4.32805e-05 after operation: -=
Diameter was nan after operation: Default construct
Diameter was 4.32805e-05 after operation: *
Diameter was 4.32805e-05 after operation: /=
Diameter was 0.00082233 after operation: -=
Diameter was nan after operation: Default construct
Diameter was 0.00082233 after operation: *
Diameter was 0.00082233 after operation: /=
Diameter was 0.0156243 after operation: -=
Diameter was nan after operation: Default construct
Diameter was 0.0156243 after operation: *
Diameter was 0.0156243 after operation: /=
Diameter was 0.296861 after operation: -=
Final value was: {8.51569e-15,1.14843e-14}
```

debug_adaptor

```
#include <boost/multiprecision/debug_adaptor.hpp>
```

```
namespace boost{ namespace multiprecision{

template <Backend>
class debug_adaptor;

}} // namespaces
```

The `debug_adaptor` type is used in conjunction with `number` and some other backend type: it acts as a thin wrapper around some other backend to class `number` and intercepts all operations on that object storing the result as a string within itself.

This type provides `numeric_limits` support whenever the template argument Backend does so.

This type is particularly useful when your debugger provides a good view of `std::string`: when this is the case multiprecision values can easily be inspected in the debugger by looking at the `debug_value` member of `debug_adaptor`. The down side of this approach is that runtimes are much slower when using this type. Set against that it can make debugging very much easier, certainly much easier than sprinkling code with `printf` statements.

When used in conjunction with the Visual C++ debugger visualisers, the value of a multiprecision type that uses this backend is displayed in the debugger just a builtin value would be, here we're inspecting a value of type `number<debug_adaptor<cpp_dec_float<50> > >`:
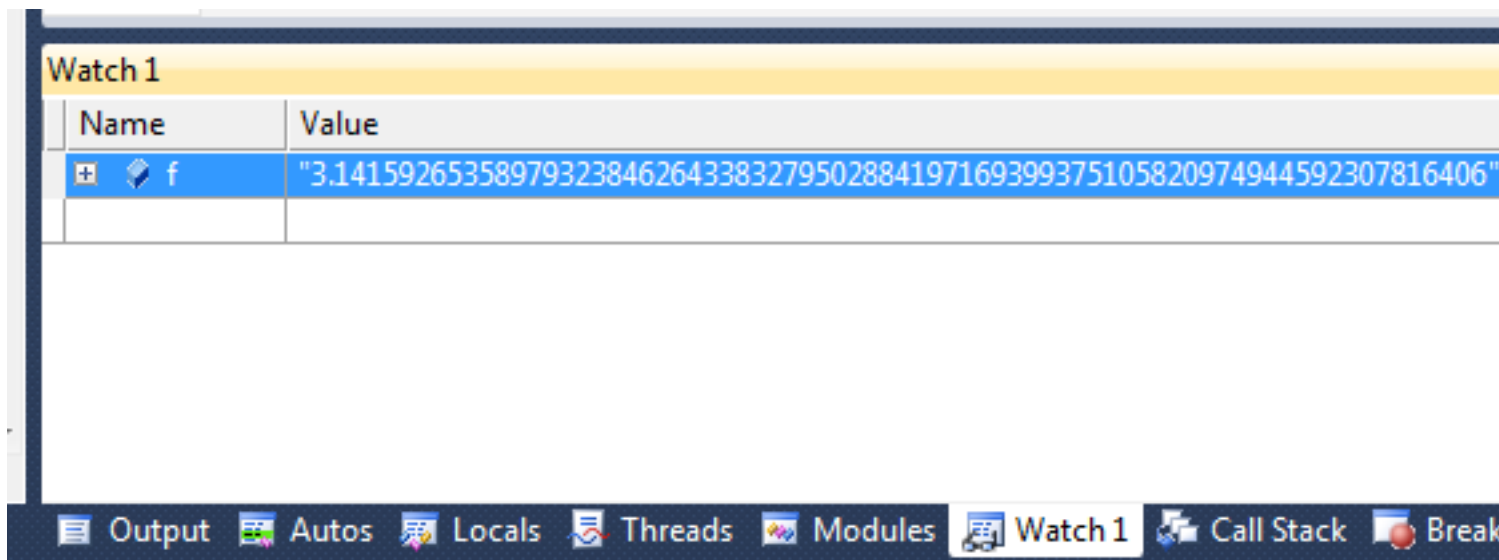
Otherwise you will need to expand out the view and look at the "debug_value" member:
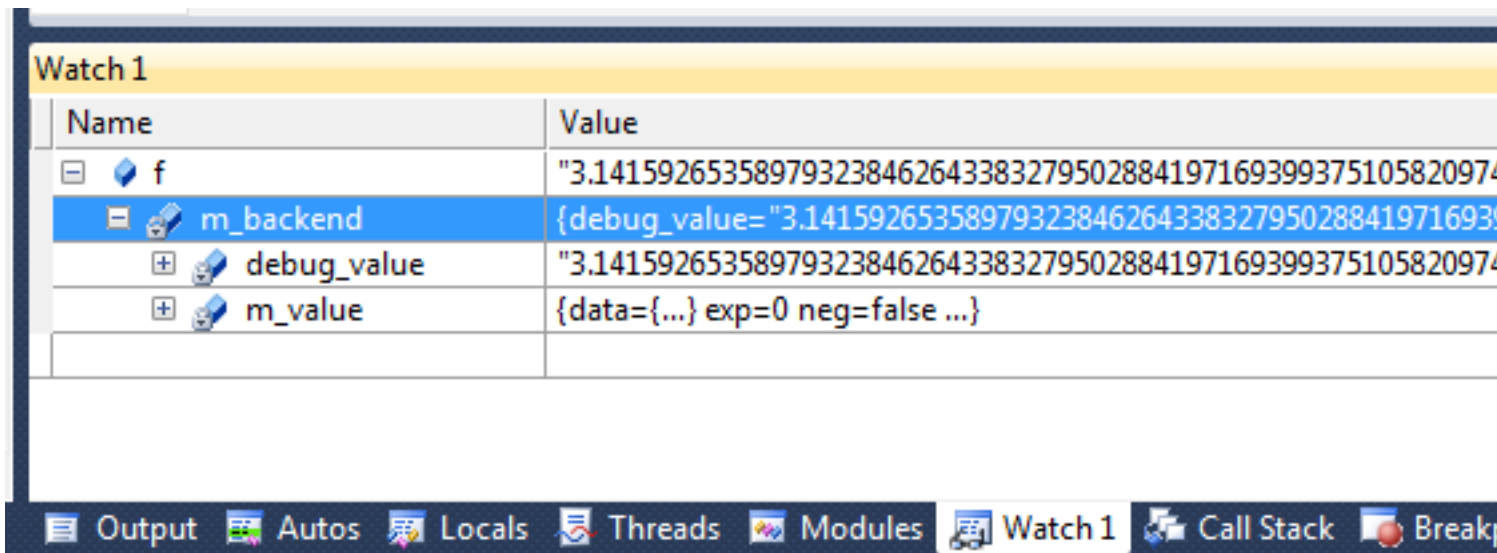
It works for all the backend types equally too, here it is inspecting a `number<debug_adaptor<gmp_rational> >`:
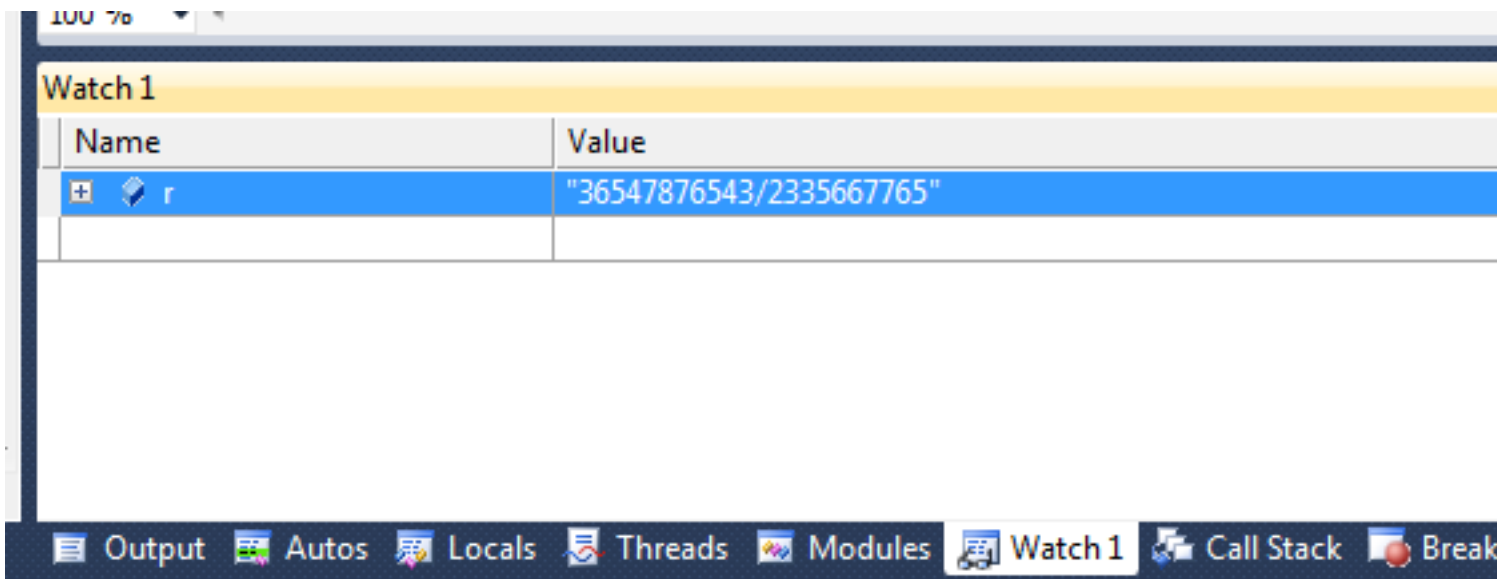
Visual C++ Debugger Visualizers

Let's face it debugger multiprecision numbers is hard - simply because we can't easily inspect the value of the numbers. Visual C++ provides a partial solution in the shape of "visualizers" which provide improved views of complex data structures, these visualizers need to be added to the `[Visualizer]` section of `autoexp.dat` located in the `Common7\Packages\Debugger` directory of your Visual Studio installation. The actual visualizer code is in the sandbox here - just cut and paste the code into your `autoexp.dat` file.

> **Note**
>
> These visualizers have only been tested with VC10, also given the ability of buggy visualizers to crash your Visual C++ debugger, make sure you back up `autoexp.dat` file before using these!!

The first visualizer provides improved views of `debug_adaptor`:
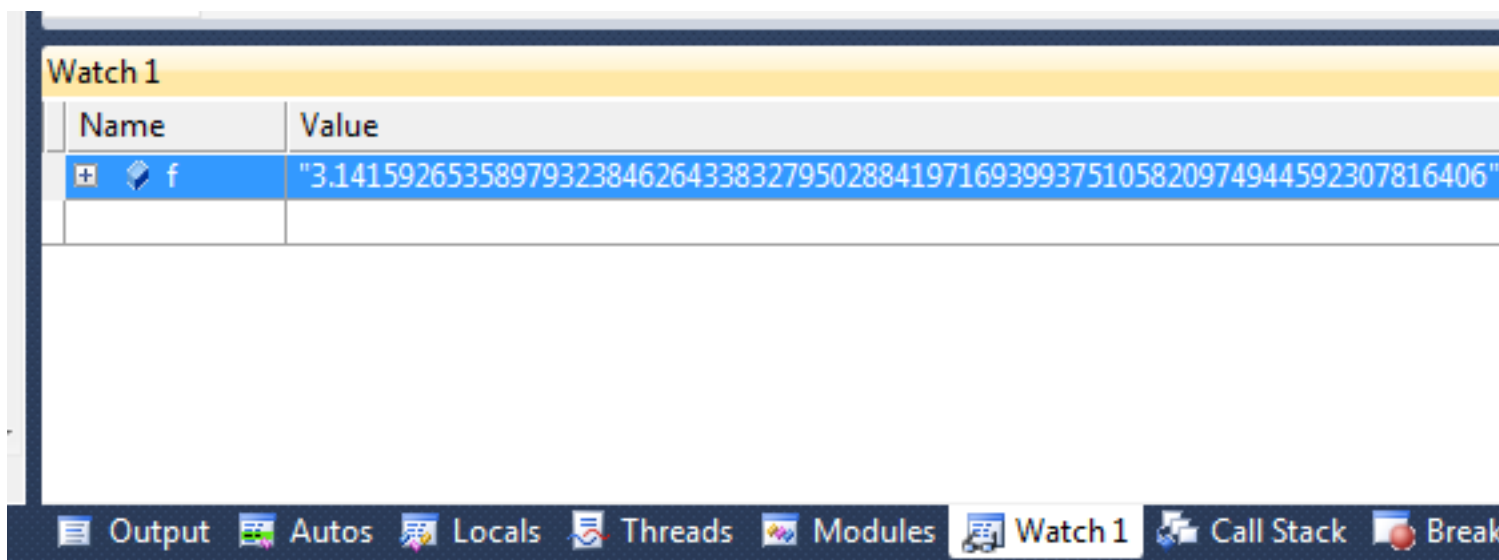
The next visualizer provides improved views of cpp_int: small numbers are displayed as actual values, while larger numbers are displayed as an array of hexadecimal parts, with the most significant part first.

Here's what it looks like for small values:
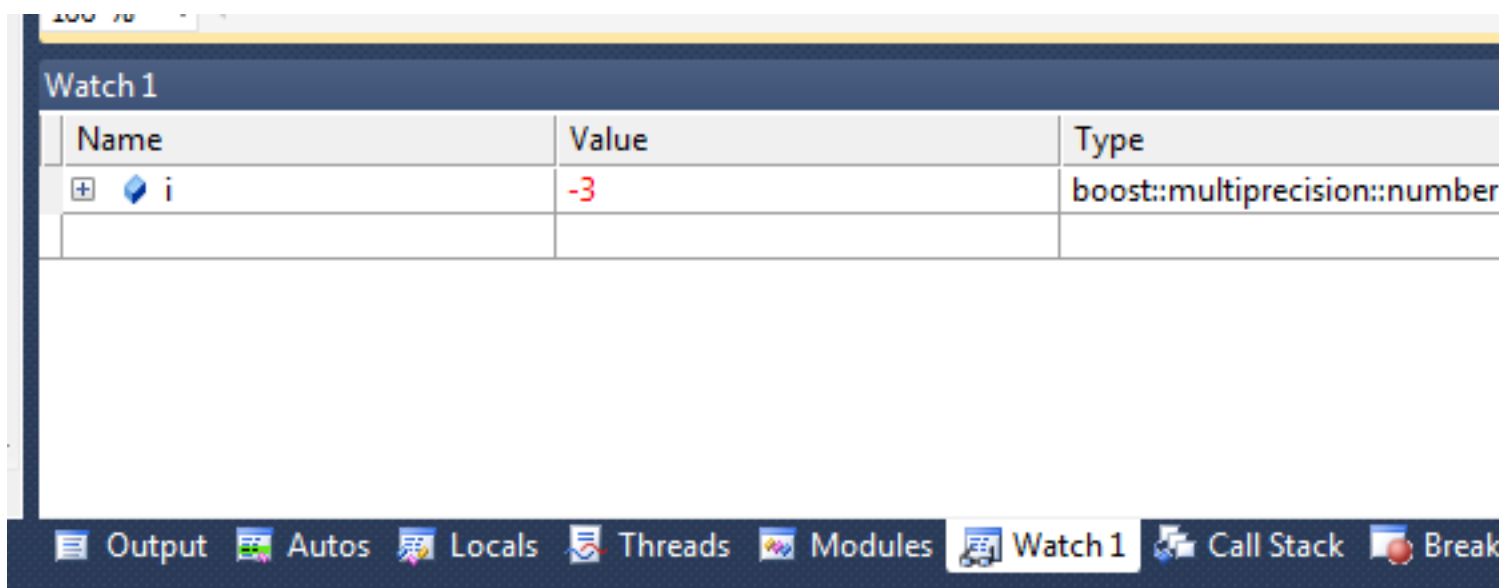
And for larger values:
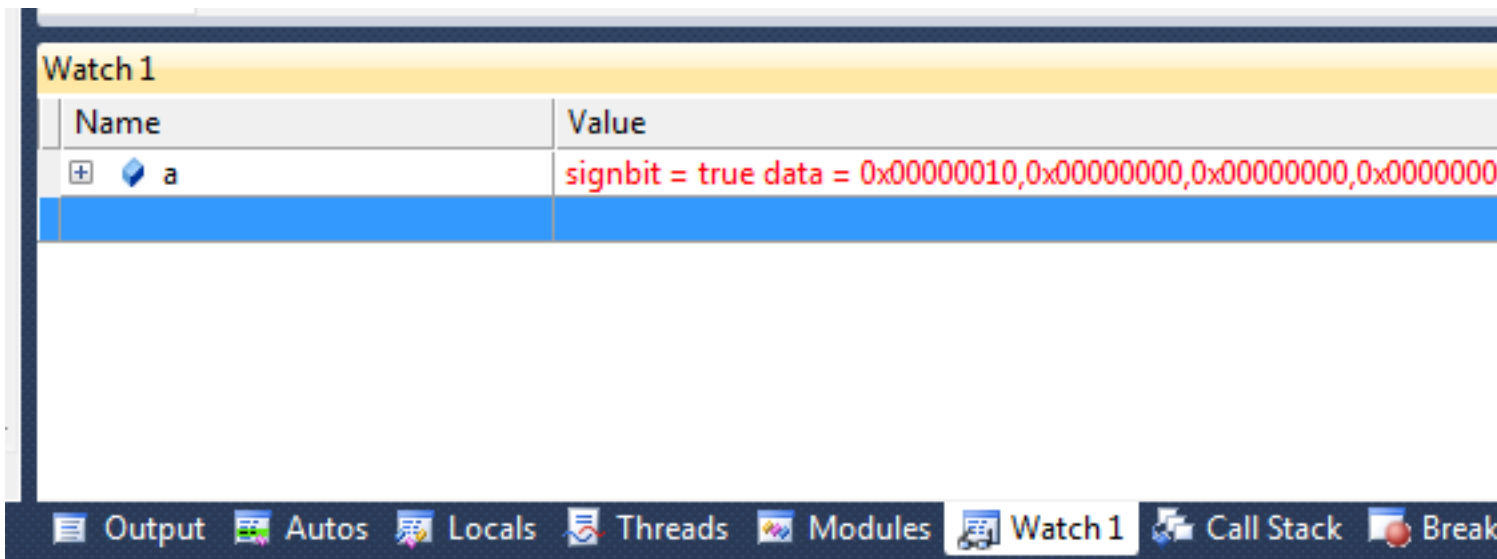
There is also a ~raw child member that lets you see the actual members of the class:
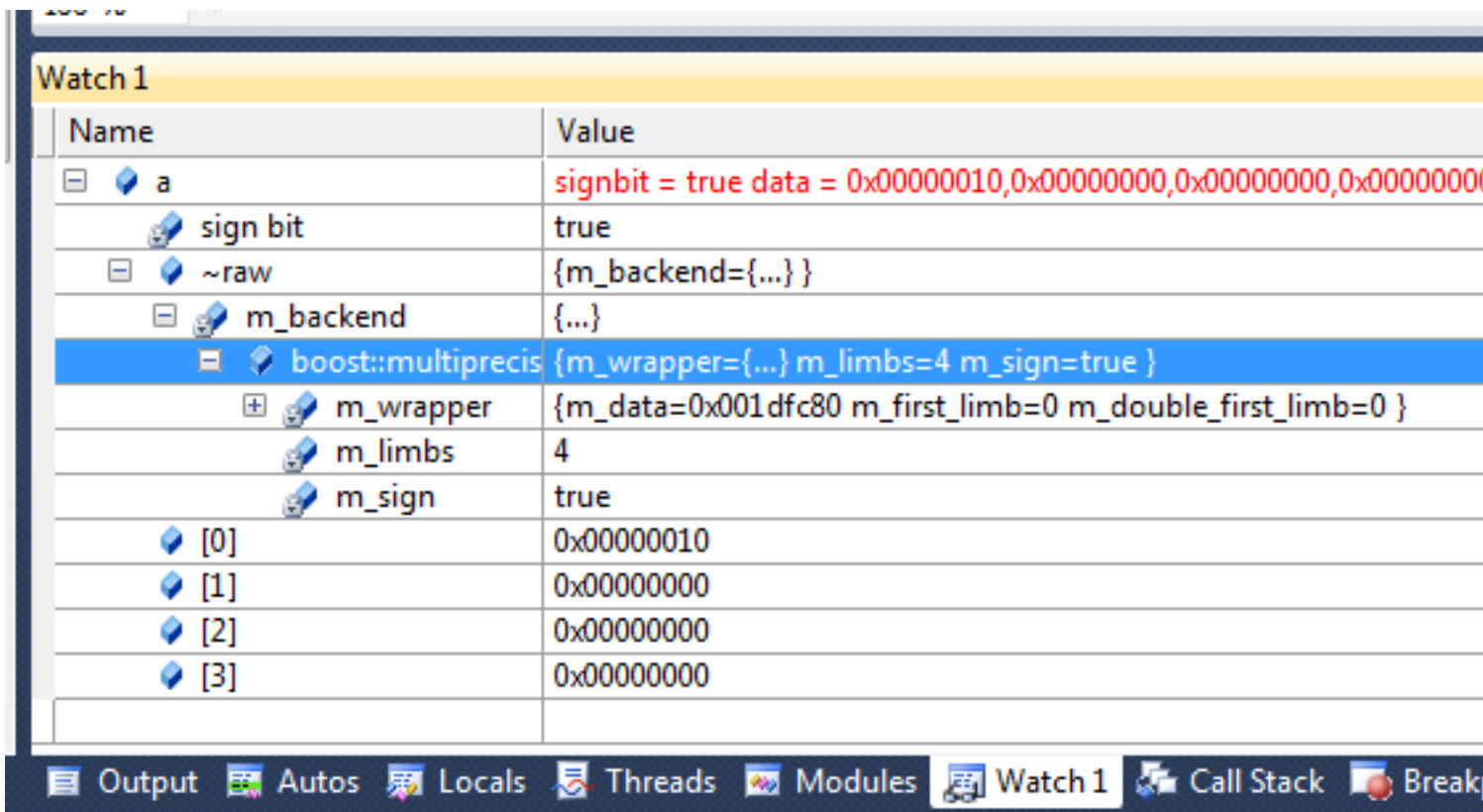
The visualizer for cpp_dec_float shows the first few digits of the value in the preview field, and the full array of digits when you expand the view. As before the ~raw child gives you access to the actual data members:

Constructing and Interconverting Between Number Types

All of the number types that are based on `number` have certain conversion rules in common. In particular:

- Any number type can be constructed (or assigned) from any builtin arithmetic type, as long as the conversion isn't lossy (for example float to int conversion):

```
cpp_dec_float_50 df(0.5);   // OK construction from double
cpp_int          i(450);    // OK constructs from signed int
cpp_int          j = 3.14;  // Error, lossy conversion.
```

- A number can be explicitly constructed from an arithmetic type, even when the conversion is lossy:

```
cpp_int          i(3.14);            // OK explicit conversion
i = static_cast<cpp_int>(3.14)       // OK explicit conversion
i.assign(3.14);                      // OK, explicit assign and avoid a temporary from the cast above
i = 3.14;                            // Error, no implicit assignment operator for lossy conversion.
cpp_int          j = 3.14;           // Error, no implicit constructor for lossy conversion.
```

- A `number` can be converted to any built in type, via the `convert_to` member function:

```
mpz_int z(2);
int i = z.template convert_to<int>(); // sets i to 2
```

Additional conversions may be supported by particular backends.

- A `number` can be converted to any built in type, via an explicit conversion operator: this functionality is only available on compilers supporting C++11's explicit conversion syntax.

```
mpz_int z(2);
int i = z;                      // Error, implicit conversion not allowed.
int j = static_cast<int>(z);    // OK explicit conversion.
```

- Any number type can be *explicitly* constructed (or assigned) from a `const char*` or a `std::string`:

```
// pi to 50 places from a string:
cpp_dec_float_50 df("3.14159265358979323846264338327950288419716939937510");
// Integer type will automatically detect "0x" and "0" prefixes and parse the string accordingly:
cpp_int          i("0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000");
// Invalid input always results in a std::runtime_error being thrown:
i = static_cast<cpp_int>("3.14");
// implicit conversions from strings are not allowed:
i = "23"; // Error, no assignment operator for implicit conversion from string
// assign member function, avoids having to create a temporary via a static_cast:
i.assign("23");  // OK
```

- Any number type will interoperate with the builtin types in arithmetic expressions as long as the conversions are not lossy:

```
// pi to 50 places from a string:
cpp_dec_float_50 df = "3.14159265358979323846264338327950288419716939937510";
// Multiply by 2 - using an integer literal here is usually more efficient
// than constructing a temporary:
df *= 2;

// You can't mix integer types with floats though:
cpp_int i = 2;
i *= 3.14;  // Error, no *= operator will be found.
```

- Any number type can be streamed to and from the C++ iostreams:

```
cpp_dec_float_50 df = "3.14159265358979323846264338327950288419716939937510";
// Now print at full precision:
std::cout << std::setprecision(std::numeric_limits<cpp_dec_float_50>::max_digits10)
   << df << std::endl
cpp_int i = 1;
i <<= 256;
// Now print in hex format with prefix:
std::cout << std::hex << std::showbase << i << std::endl;
```

- Interconversions between number types of the same family are allowed and are implicit conversions if no loss of precision is involved, and explicit if it is:

```
int128_t      i128 = 0;
int266_t      i256 = i128;  // OK implicit widening conversion
i128_t            = i256;  // Error, no assignment operator found, narrowing conversion is explicit
i128_t            = static_cast<int128_t>(i256); // OK, explicit narrowing conversion

mpz_int       z    = 0;
mpf_float     f    = z;     // OK, GMP handles this conversion natively, and it's not lossy and ↵
therefore implicit

mpf_float_50 f50  = 2;
f                 = f50;   // OK, conversion from fixed to variable precision, f will have 50 di↵
gits precision.
f50               = f;     // Error, conversion from variable to fixed precision is potentially ↵
lossy, explicit cast required.
```

- Some interconversions between number types are completely generic, and are always available, albeit the conversions are always *explicit*:

```cpp
cpp_int cppi(2);
// We can always convert between numbers of the same category -
// int to int, rational to rational, or float to float, so this is OK
// as long as we use an explicit conversion:
mpz_int z(cppi);
// We can always promote from int to rational, int to float, or rational to float:
cpp_rational     cppr(cppi);   // OK, int to rational
cpp_dec_float_50 df(cppi);     // OK, int to float
df               = static_cast<cpp_dec_float_50>(cppr);  // OK, explicit rational to float ↵
conversion
// However narrowing and/or implicit conversions always fail:
cppi             =  df;    // Compiler error, conversion not allowed
```

- Other interconversions may be allowed as special cases, whenever the backend allows it:

```cpp
mpf_t     m;              // Native GMP type.
mpf_init_set_ui(m, 0); // set to a value;
mpf_float i(m);          // copies the value of the native type.
```

More information on what additional types a backend supports conversions from are given in the tutorial for each backend. The converting constructor will be implicit if the backend's converting constructor is also implicit, and explicit if the backends converting constructor is also explicit.

Generating Random Numbers

Random numbers are generated in conjunction with Boost.Random. However, since Boost.Random is unaware of arbitrary precision numbers, it's necessary to include the header:

```cpp
#include <boost/multiprecision/random.hpp>
```

In order to act as a bridge between the two libraries.

Integers with *N* random bits are generated using `independent_bits_engine`:

```cpp
#include <boost/multiprecision/gmp.hpp>
#include <boost/multiprecision/random.hpp>

using namespace boost::multiprecision;
using namespace boost::random;

//
// Declare our random number generator type, the underlying generator
// is the Mersenne twister mt19937 engine, and 256 bits are generated:
//
typedef independent_bits_engine<mt19937, 256, mpz_int> generator_type;
generator_type gen;
//
// Generate some values:
//
std::cout << std::hex << std::showbase;
for(unsigned i = 0; i < 10; ++i)
   std::cout << gen() << std::endl;
```

Alternatively we can generate integers in a given range using `uniform_int_distribution`, this will invoke the underlying engine multiple times to build up the required number of bits in the result:

```cpp
#include <boost/multiprecision/gmp.hpp>
#include <boost/multiprecision/random.hpp>

using namespace boost::multiprecision;
using namespace boost::random;


//
// Generate integers in a given range using uniform_int,
// the underlying generator is invoked multiple times
// to generate enough bits:
//
mt19937 mt;
uniform_int_distribution<mpz_int> ui(0, mpz_int(1) << 256);
//
// Generate the numbers:
//
std::cout << std::hex << std::showbase;
for(unsigned i = 0; i < 10; ++i)
   std::cout << ui(mt) << std::endl;
```

Floating point values in [0,1) are generated using `uniform_01`, the trick here is to ensure that the underlying generator produces as many random bits as there are digits in the floating point type. As above `independent_bits_engine` can be used for this purpose, note that we also have to convert decimal digits (in the floating point type) to bits (in the random number generator):

```cpp
#include <boost/multiprecision/gmp.hpp>
#include <boost/multiprecision/random.hpp>

using namespace boost::multiprecision;
using namespace boost::random;
//
// We need an underlying generator with at least as many bits as the
// floating point type to generate numbers in [0, 1) with all the bits
// in the floating point type randomly filled:
//
uniform_01<mpf_float_50> uf;
independent_bits_engine<mt19937, 50L*1000L/301L, mpz_int> gen;
//
// Generate the values:
//
std::cout << std::setprecision(50);
for(unsigned i = 0; i < 20; ++i)
   std::cout << uf(gen) << std::endl;
```

Finally, we can modify the above example to produce numbers distributed according to some distribution:

```
#include <boost/multiprecision/gmp.hpp>
#include <boost/multiprecision/random.hpp>

using namespace boost::multiprecision;
using namespace boost::random;
//
// We can repeat the above example, with other distributions:
//
uniform_real_distribution<mpf_float_50> ur(-20, 20);
gamma_distribution<mpf_float_50> gd(20);
independent_bits_engine<mt19937, 50L*1000L/301L, mpz_int> gen;
//
// Generate some values:
//
std::cout << std::setprecision(50);
for(unsigned i = 0; i < 20; ++i)
   std::cout << ur(gen) << std::endl;
for(unsigned i = 0; i < 20; ++i)
   std::cout << gd(gen) << std::endl;
```

Primality Testing

The library implements a Miller-Rabin test for primality:

```
#include <boost/multiprecision/miller_rabin.hpp>

template <class Backend, expression_template_option ExpressionTemplates, class Engine>
bool miller_rabin_test(const number<Backend, ExpressionTemplates>& n, unsigned trials, En⏎
gine& gen);

template <class Backend, expression_template_option ExpressionTemplates, class Engine>
bool miller_rabin_test(const number<Backend, ExpressionTemplates>& n, unsigned trials);
```

These functions perform a Miller-Rabin test for primality, if the result is `false` then *n* is definitely composite, while if the result is `true` then *n* is prime with probability *0.25^trials*. The algorithm used performs some trial divisions to exclude small prime factors, does one Fermat test to exclude many more composites, and then uses the Miller-Rabin algorithm straight out of Knuth Vol 2, which recommends 25 trials for a pretty strong likelihood that *n* is prime.

The third optional argument is for a Uniform Random Number Generator from Boost.Random. When not provided the `mt19937` generator is used. Note that when producing random primes then you should probably use a different random number generator to produce candidate prime numbers for testing, than is used internally by `miller_rabin_test` for determining whether the value is prime. It also helps of course to seed the generators with some source of randomness.

The following example searches for a prime `p` for which `(p-1)/2` is also probably prime:

```
#include <boost/multiprecision/cpp_int.hpp>
#include <boost/multiprecision/miller_rabin.hpp>
#include <iostream>
#include <iomanip>

int main()
{
   using namespace boost::random;
   using namespace boost::multiprecision;

   typedef cpp_int int_type;
   mt11213b base_gen(clock());
   independent_bits_engine<mt11213b, 256, int_type> gen(base_gen);
   //
   // We must use a different generator for the tests and number generation, otherwise
   // we get false positives.
   //
   mt19937 gen2(clock());

   for(unsigned i = 0; i < 100000; ++i)
   {
      int_type n = gen();
      if(miller_rabin_test(n, 25, gen2))
      {
         // Value n is probably prime, see if (n-1)/2 is also prime:
         std::cout << "We have a probable prime with value: " << std::hex << std::show↵
base << n << std::endl;
         if(miller_rabin_test((n-1)/2, 25, gen2))
         {
            std::cout << "We have a safe prime with value: " << std::hex << std::show↵
base << n << std::endl;
            return 0;
         }
      }
   }
   std::cout << "Ooops, no safe primes were found" << std::endl;
   return 1;
}
```

Literal Types and `constexpr` Support

> **Note**
>
> The features described in this section make heavy use of C++11 language features, currently (as of May 2013) only GCC-4.7 and later, and Clang 3.3 and later have the support required to make these features work.

There is limited support for `constexpr` and user-defined literals in the library, currently the `number` front end supports `constexpr` on default construction and all forwarding constructors, but not on any of the non-member operators. So if some type `B` is a literal type, then `number<B>` is also a literal type, and you will be able to compile-time-construct such a type from any literal that `B` is compile-time-constructible from. However, you will not be able to perform compile-time arithmetic on such types.

Currently the only backend type provided by the library that is also a literal type are instantiations of `cpp_int_backend` where the Allocator parameter is type `void`, and the Checked parameter is `boost::multiprecision::unchecked`.

For example:

```
using namespace boost::multiprecision;

constexpr int128_t          i = 0;      // OK, fixed precision int128_t has no allocator.
constexpr uint1024_t        j = 0xFFFFFFFF00000000uLL;  // OK, fixed precision uint1024_t has ↵
no allocator.

constexpr checked_uint128_t  k = -1; // Error, checked type is not a literal type as we need ↵
runtime error checking.
constexpr cpp_int            l = 2;  // Error, type is not a literal as it performs memory man↵
agement.
```

There is also limited support for user defined-literals - these are limited to unchecked, fixed precision `cpp_int`'s which are specified in hexadecimal notation. The suffixes supported are:

Suffix	Meaning
_cppi	Specifies a value of type: number<cpp_int_backend<N,N,signed_magnitude,unchecked,void> >, where N is chosen to contain just enough digits to hold the number specified.
_cppui	Specifies a value of type: number<cpp_int_backend<N,N,unsigned_magnitude,unchecked,void> >, where N is chosen to contain just enough digits to hold the number specified.
_cppi*N*	Specifies a value of type number<cpp_int_backend<N,N,signed_magnitude,unchecked,void> >.
_cppui*N*	Specifies a value of type number<cpp_int_backend<N,N,signed_magnitude,unchecked,void> >.

In each case, use of these suffixes with hexadecimal values produces a `constexpr` result.

Examples:

```
//
// Any use of user defined literals requires that we import the literal-operators
// into current scope first:
using namespace boost::multiprecision::literals;
//
// To keep things simple in the example, we'll make our types used visible to this scope as well:
using namespace boost::multiprecision;
//
// The value zero as a number<cpp_int_backend<4,4,signed_magnitude,unchecked,void> >:
constexpr auto a = 0x0_cppi;
// The type of each constant has 4 bits per hexadecimal digit,
// so this is of type uint256_t (ie number<cpp_int_backend<256,256,unsigned_magnitude,un↵
checked,void> >):
constexpr auto b = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF_cppui;
//
// Smaller values can be assigned to larger values:
int256_t c = 0x1234_cppi; // OK
//
// However, this does not currently work in constexpr contexts:
constexpr int256_t d = 0x1_cppi; // Compiler error
//
// Constants can be padded out with leading zeros to generate wider types:
constexpr uint256_t e = 0x00000000000000000000000000000000000000000000FFFFFFFFFFFFFFFFFFFF_cp↵
pui; // OK
//
// However, specific width types are best produced with specific-width suffixes,
// ones supported by default are `_cpp[u]i128`, `_cpp[u]i256`, `_cpp[u]i512`, `_cpp[u]i1024`.
//
constexpr int128_t f = 0x1234_cppi128; // OK, always produces an int128_t as the result.
constex↵
pr uint1024_t g = 0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbcccccccccccccccccccccc_cp↵
pui1024;
//
// If other specific width types are required, then there is a macro for generating the operators
// for these.  The macro can be used at namespace scope only:
//
BOOST_MP_DEFINE_SIZED_CPP_INT_LITERAL(2048);
//
// Now we can create 2048-bit literals as well:
constexpr auto h = 0xff_cppi2048; // h is of type number<cpp_int_backend<2048,2048,signed_mag↵
nitude,unchecked,void> >
//
// Finally negative values are handled via the unary minus operator:
//
constexpr int1024_t i = -0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF_cp↵
pui1024;
//
// Which means this also works:
constexpr int1024_t j = -g;    // OK: unary minus operator is constexpr.
```

Rounding Rules for Conversions

As a general rule, all conversions between unrelated types are performed using basic arithmetic operations, therefore conversions are either exact, or follow the same rounding rules as arithmetic for the type in question.

The following table summarises the situation for conversions from native types:

Backend	Rounding Rules
cpp_int	Conversions from integer types are exact if the target has sufficient precision, otherwise they truncate to the first 2^MaxBits bits (modulo arithmetic). Conversions from floating point types are truncating to the nearest integer.
gmp_int	Conversions are performed by the GMP library except for conversion from `long double` which is truncating.
tom_int	Conversions from floating point types are truncating, all others are performed by libtommath and are exact.
gmp_float	Conversions are performed by the GMP library except for conversion from `long double` which should be exact provided the target type has as much precision as a `long double`.
mpfr_float	All conversions are performed by the underlying MPFR library.
cpp_dec_float	All conversions are performed using basic arithmetic operations and are truncating.
gmp_rational	See gmp_int
cpp_rational	See cpp_int
tommath_rational	See tom_int

Mixed Precision Arithmetic

Mixed precision arithmetic is fully supported by the library.

There are two different forms:

- Where the operands are of different precision.

- Where the operands are of the same precision, but yield a higher precision result.

Mixing Operands of Differing Precision

If the arguments to a binary operator are of different precision, then the operation is allowed as long as there is an unambiguous implicit conversion from one argument type to the other. In all cases the arithmetic is performed "as if" the lower precision type is promoted to the higher precision type before applying the operator. However, particular backends may optimise this and avoid actually creating a temporary if they are able to do so.

For example:

```
mpfr_float_50         a(2), b;
mpfr_float_100        c(3), d;
static_mpfr_float_50  e(5), f;
mpz_int               i(20);

d = a * c;  // OK, result of operand is an mpfr_float_100.
b = a * c;  // Error, can't convert the result to an mpfr_float_50 as it will lose digits.
f = a * e;  // Error, operator is ambiguous, result could be of either type.
f = e * i;  // OK, unambiguous conversion from mpz_int to static_mpfr_float_50
```

Operands of the Same Precision

Sometimes you want to apply an operator to two arguments of the same precision in such a way as to obtain a result of higher precision. The most common situation occurs with fixed precision integers, where you want to multiply two N-bit numbers to obtain a 2N-bit result. This is supported in this library by the following free functions:

```
template <class ResultType, class Source1 class Source2>
ResultType& add(ResultType& result, const Source1& a, const Source2& b);

template <class ResultType, class Source1 class Source2>
ResultType& subtract(ResultType& result, const Source1& a, const Source2& b);

template <class ResultType, class Source1 class Source2>
ResultType& multiply(ResultType& result, const Source1& a, const Source2& b);
```

These functions apply the named operator to the arguments *a* and *b* and store the result in *result*, returning *result*. In all cases they behave "as if" arguments *a* and *b* were first promoted to type `ResultType` before applying the operator, though particular backends may well avoid that step by way of an optimization.

The type `ResultType` must be an instance of class `number`, and the types `Source1` and `Source2` may be either instances of class `number` or native integer types. The latter is an optimization that allows arithmetic to be performed on native integer types producing an extended precision result.

For example:

```
#include <boost/multiprecision/cpp_int.hpp>

using namespace boost::multiprecision;

boost::uint64_t i = (std::numeric_limits<boost::uint64_t>::max)();
boost::uint64_t j = 1;

uint128_t ui128;
uint256_t ui256;
//
// Start by performing arithmetic on 64-bit integers to yield 128-bit results:
//
std::cout << std::hex << std::showbase << i << std::endl;
std::cout << std::hex << std::showbase << add(ui128, i, j) << std::endl;
std::cout << std::hex << std::showbase << multiply(ui128, i, i) << std::endl;
//
// The try squaring a 128-bit integer to yield a 256-bit result:
//
ui128 = (std::numeric_limits<uint128_t>::max)();
std::cout << std::hex << std::showbase << multiply(ui256, ui128, ui128) << std::endl;
```

Produces the output:

```
0xffffffffffffffff
0x10000000000000000
0xFFFFFFFFFFFFFFFE0000000000000001
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE0000000000000000000000000000000001
```

Backends With Optimized Mixed Precision Arithmetic

The following backends have at least some direct support for mixed precision arithmetic, and therefore avoid creating unnecessary temporaries when using the interfaces above. Therefore when using these types it's more efficient to use mixed precision arithmetic, than it is to explicitly cast the operands to the result type:

mpfr_float, gmp_float, cpp_int.

Reference

number

Synopsis

```cpp
namespace boost{ namespace multiprecision{

enum expression_template_option { et_on = 1, et_off = 0 };

template <class Backend> struct expression_template_default
{ static const expression_template_option value = et_on; };

template <class Backend, expression_template_option ExpressionTemplates = expression_template_de↵
fault<Backend>::value>
class number
{
   number();
   number(see-below);
   number& operator=(see-below);
   number& assign(see-below);

   // Member operators
   number& operator+=(const see-below&);
   number& operator-=(const see-below&);
   number& operator*=(const see-below&);
   number& operator/=(const see-below&);
   number& operator++();
   number& operator--();
   number  operator++(int);
   number  operator--(int);

   number& operator%=(const see-below&);
   number& operator&=(const see-below&);
   number& operator|=(const see-below&);
   number& operator^=(const see-below&);
   number& operator<<=(const integer-type&);
   number& operator>>=(const integer-type&);

   // Use in Boolean context:
   operator convertible-to-bool-type()const;
   // swap:
   void swap(number& other);
   // Sign:
   bool is_zero()const;
   int sign()const;
   // string conversion:
   std::string str()const;
   // Generic conversion mechanism
   template <class T>
   T convert_to()const;
   template <class T>
   explicit operator T ()const;
   // precision control:
   static unsigned default_precision();
   static void default_precision(unsigned digits10);
   unsigned precision()const;
   void precision(unsigned digits10);
   // Comparison:
   int compare(const number<Backend>& o)const;
   template <class V>
```

```
    typename enable_if<is_convertible<V, number<Backend, ExpressionTemplates> >, int>::type
        compare(const V& o)const;
    // Access to the underlying implementation:
    Backend& backend();
    const Backend& backend()const;
};

// Non member operators:
unmentionable-expression-template-type operator+(const see-below&);
unmentionable-expression-template-type operator-(const see-below&);
unmentionable-expression-template-type operator+(const see-below&, const see-below&);
unmentionable-expression-template-type operator-(const see-below&, const see-below&);
unmentionable-expression-template-type operator*(const see-below&, const see-below&);
unmentionable-expression-template-type operator/(const see-below&, const see-below&);
// Integer only operations:
unmentionable-expression-template-type operator%(const see-below&, const see-below&);
unmentionable-expression-template-type operator&(const see-below&, const see-below&);
unmentionable-expression-template-type operator|(const see-below&, const see-below&);
unmentionable-expression-template-type operator^(const see-below&, const see-below&);
unmentionable-expression-template-type operator<<(const see-below&, const integer-type&);
unmentionable-expression-template-type operator>>(const see-below&, const integer-type&);
// Comparison operators:
bool operator==(const see-below&, const see-below&);
bool operator!=(const see-below&, const see-below&);
bool operator< (const see-below&, const see-below&);
bool operator> (const see-below&, const see-below&);
bool operator<=(const see-below&, const see-below&);
bool operator>=(const see-below&, const see-below&);

// Swap:
template <class Backend, expression_template_option ExpressionTemplates>
void swap(number<Backend, ExpressionTemplates>& a, number<Backend, ExpressionTemplates>& b);

// iostream support:
template <class Backend, expression_template_option ExpressionTemplates>
std::ostream& operator << (std::ostream& os, const number<Backend, ExpressionTemplates>& r);
std::ostream& operator << (std::ostream& os, const unmentionable-expression-template-type& r);
template <class Backend, expression_template_option ExpressionTemplates>
std::istream& operator >> (std::istream& is, number<Backend, ExpressionTemplates>& r);

// Arithmetic with a higher precision result:
template <class ResultType, class Source1 class Source2>
ResultType& add(ResultType& result, const Source1& a, const Source2& b);
template <class ResultType, class Source1 class Source2>
ResultType& subtract(ResultType& result, const Source1& a, const Source2& b);
template <class ResultType, class Source1 class Source2>
ResultType& multiply(ResultType& result, const Source1& a, const Source2& b);

// Non-member function standard library support:
unmentionable-expression-template-type    abs    (const number-or-expression-template-type&);
unmentionable-expression-template-type    fabs   (const number-or-expression-template-type&);
unmentionable-expression-template-type    sqrt   (const number-or-expression-template-type&);
unmentionable-expression-template-type    floor  (const number-or-expression-template-type&);
unmentionable-expression-template-type    ceil   (const number-or-expression-template-type&);
unmentionable-expression-template-type    trunc  (const number-or-expression-template-type&);
unmentionable-expression-template-type    itrunc (const number-or-expression-template-type&);
unmentionable-expression-template-type    ltrunc (const number-or-expression-template-type&);
unmentionable-expression-template-type    lltrunc(const number-or-expression-template-type&);
unmentionable-expression-template-type    round  (const number-or-expression-template-type&);
unmentionable-expression-template-type    iround (const number-or-expression-template-type&);
unmentionable-expression-template-type    lround (const number-or-expression-template-type&);
unmentionable-expression-template-type    llround(const number-or-expression-template-type&);
unmentionable-expression-template-type    exp    (const number-or-expression-template-type&);
```

```
unmentionable-expression-template-type     log    (const number-or-expression-template-type&);
unmentionable-expression-template-type     log10   (const number-or-expression-template-type&);
unmentionable-expression-template-type     cos    (const number-or-expression-template-type&);
unmentionable-expression-template-type     sin    (const number-or-expression-template-type&);
unmentionable-expression-template-type     tan    (const number-or-expression-template-type&);
unmentionable-expression-template-type     acos   (const number-or-expression-template-type&);
unmentionable-expression-template-type     asin   (const number-or-expression-template-type&);
unmentionable-expression-template-type     atan   (const number-or-expression-template-type&);
unmentionable-expression-template-type     cosh   (const number-or-expression-template-type&);
unmentionable-expression-template-type     sinh   (const number-or-expression-template-type&);
unmentionable-expression-template-type     tanh   (const number-or-expression-template-type&);

unmentionable-expression-template-type      ldexp (const number-or-expression-template-type&, int);
unmentionable-expression-template-type      frexp (const number-or-expression-template-type&, int*);
unmentionable-expression-template-type      pow   (const number-or-expression-template-
type&, const number-or-expression-template-type&);
unmentionable-expression-template-type      fmod  (const number-or-expression-template-
type&, const number-or-expression-template-type&);
unmentionable-expression-template-type      atan2 (const number-or-expression-template-
type&, const number-or-expression-template-type&);

// Traits support:
template <class T>
struct component_type;
template <class T>
struct number_category;
template <class T>
struct is_number;
template <class T>
struct is_number_expression;

// Integer specific functions:
unmentionable-expression-template-type    pow(const number-or-expression-template-type&, unsigned);
unmentionable-expression-template-type    powm(const number-or-expression-template-
type& b, const number-or-expression-template-type& p, const number-or-expression-template-type& m);
template <class Backend, expression_template_option ExpressionTemplates>
void divide_qr(const number-or-expression-template-type& x, const number-or-expression-template-
type& y,
            number<Backend, ExpressionTemplates>& q, number<Backend, ExpressionTemplates>& r);
template <class Integer>
Integer integer_modulus(const number-or-expression-template-type& x, Integer val);
unsigned lsb(const number-or-expression-template-type& x);
template <class Backend, class ExpressionTemplates>
bool bit_test(const number<Backend, ExpressionTemplates>& val, unsigned index);
template <class Backend, class ExpressionTemplates>
number<Backend, ExpressionTemplates>& bit_set(number<Backend, ExpressionTemplates>& val, un↵
signed index);
template <class Backend, class ExpressionTemplates>
number<Backend, ExpressionTemplates>& bit_unset(number<Backend, ExpressionTemplates>& val, un↵
signed index);
template <class Backend, class ExpressionTemplates>
number<Backend, ExpressionTemplates>& bit_flip(number<Backend, ExpressionTemplates>& val, un↵
signed index);
template <class Engine>
bool miller_rabin_test(const number-or-expression-template-type& n, unsigned trials, Engine& gen);
bool miller_rabin_test(const number-or-expression-template-type& n, unsigned trials);

// Rational number support:
typename component_type<number-or-expression-template-type>::type numerator  (const number-or-
expression-template-type&);
typename component_type<number-or-expression-template-type>::type denominator(const number-or-
expression-template-type&);
```

```
}} // namespaces

namespace boost{ namespace math{

// Boost.Math interoperability functions:
int                                            fpclassify      (const number-or-expression-
template-type&, int);
bool                                           isfinite        (const number-or-expression-
template-type&, int);
bool                                           isnan           (const number-or-expression-
template-type&, int);
bool                                           isinf           (const number-or-expression-
template-type&, int);
bool                                           isnormal        (const number-or-expression-
template-type&, int);

}} // namespaces

// numeric_limits support:
namespace std{

template <class Backend, expression_template_option ExpressionTemplates>
struct numeric_limits<boost::multiprecision<Backend, ExpressionTemplates> >
{
   /* Usual members here */
};

}
```

Description

```
enum expression_template_option { et_on = 1, et_off = 0 };
```

This enumerated type is used to specify whether expression templates are turned on (et_on) or turned off (et_off).

```
template <class Backend> struct expression_template_default
{ static const expression_template_option value = et_on; };
```

This traits class specifies the default expression template option to be used with a particular Backend type. It defaults to `et_on`.

```
template <class Backend, expression_template_option ExpressionTemplates = expression_template_de↵
fault<Backend>::value>
class number;
```

Class `number` has two template arguments:

Backend   The actual arithmetic back-end that does all the work.

ExpressionTemplates   A Boolean value: when `et_on`, then expression templates are enabled, otherwise when set to `et_off`
                        they are disabled. The default for this parameter is computed via the traits class `expression_tem-`
                        `plate_default` whose member `value` defaults to `et_on` unless the the traits class is specialized
                        for a particular backend.

```
number();
number(see-below);
number& operator=(see-below);
number& assign(see-below);
```

Type `number` is default constructible, and both copy constructible and assignable from:

- Itself.

- An expression template which is the result of one of the arithmetic operators.

- Any builtin arithmetic type, as long as the result would not be lossy (for example float to integer conversion).

- Any type that the Backend is implicitly constructible or assignable from.

- An rvalue reference to another `number`. Move-semantics are used for construction if the backend also supports rvalue reference construction. In the case of assignment, move semantics are always supported when the argument is an rvalue reference irrespective of the backend.

- Any type in the same family, as long as no loss of precision is involved. For example from `int128_t` to `int256_t`, or `cpp_dec_float_50` to `cpp_dec_float_100`.

Type `number` is explicitly constructible from:

- Any type mentioned above.

- A `std::string` or any type which is convertible to `const char*`.

- Any arithmetic type (including those that would result in lossy conversions).

- Any type in the same family, including those that result in loss of precision.

- Any type that the Backend is explicitly constructible from.

- Any pair of types for which a generic interconversion exists: that is from integer to integer, integer to rational, integer to float, rational to rational, rational to float, or float to float.

The assign member function is available for any type for which an explicit converting constructor exists. It is intended to be used where a temporary generated from an explicit assignment would be expensive, for example:

```
mpfr_float_50    f50;
mpfr_float_100   f100;

f50 = static_cast<mpfr_float_50>(f100);  // explicit cast create a temporary
f50.assign(f100);                        // explicit call to assign create no temporary
```

In addition, if the type has multiple components (for example rational or complex number types), then there is a two argument constructor:

```
number(arg1, arg2);
```

Where the two args must either be arithmetic types, or types that are convertible to the two components of `this`.

```
number& operator+=(const see-below&);
number& operator-=(const see-below&);
number& operator*=(const see-below&);
number& operator/=(const see-below&);
number& operator++();
number& operator--();
number  operator++(int);
number  operator--(int);
// Integer only operations:
number& operator%=(const see-below&);
number& operator&=(const see-below&);
number& operator|=(const see-below&);
number& operator^=(const see-below&);
number& operator<<=(const integer-type&);
number& operator>>=(const integer-type&);
```

These operators all take their usual arithmetic meanings.

The arguments to these operators is either:

- Another `number<Backend, ExpressionTemplates>`.

- An expression template derived from `number<Backend>`.

- Any type implicitly convertible to `number<Backend, ExpressionTemplates>`, including some other instance of class `number`.

For the left and right shift operations, the argument must be a builtin integer type with a positive value (negative values result in a `std::runtime_error` being thrown).

```
operator convertible-to-bool-type()const;
```

Returns an *unmentionable-type* that is usable in Boolean contexts (this allows `number` to be used in any Boolean context - if statements, conditional statements, or as an argument to a logical operator - without type `number` being convertible to type `bool`.

This operator also enables the use of `number` with any of the following operators: `!`, `||`, `&&` and `?:`.

```
void swap(number& other);
```

Swaps `*this` with `other`.

```
bool is_zero()const;
```

Returns `true` is `*this` is zero, otherwise `false`.

```
int sign()const;
```

Returns a value less than zero if `*this` is negative, a value greater than zero if `*this` is positive, and zero if `*this` is zero.

```
std::string str(unsigned precision, bool scientific = true)const;
```

Returns the number formatted as a string, with at least *precision* digits, and in scientific format if *scientific* is true.

```
template <class T>
T convert_to()const;

template <class T>
explicit operator T ()const;
```

Provides a generic conversion mechanism to convert *this to type T. Type T may be any arithmetic type. Optionally other types may also be supported by specific Backend types.

```
static unsigned default_precision();
static void default_precision(unsigned digits10);
unsigned precision()const;
void precision(unsigned digits10);
```

These functions are only available if the Backend template parameter supports runtime changes to precision. They get and set the default precision and the precision of *this respectively.

```
int compare(const number<Backend, ExpressionTemplates>& o)const;
template <class V>
typename enable_if<is_convertible<V, number<Backend, ExpressionTemplates> >, int>::type
    compare(const V& other)const;
```

Returns:

- A value less that 0 for *this < other

- A value greater that 0 for *this > other

- Zero for *this == other

```
Backend& backend();
const Backend& backend()const;
```

Returns the underlying back-end instance used by *this.

Non-member operators

```
// Non member operators:
unmentionable-expression-template-type operator+(const see-below&);
unmentionable-expression-template-type operator-(const see-below&);
unmentionable-expression-template-type operator+(const see-below&, const see-below&);
unmentionable-expression-template-type operator-(const see-below&, const see-below&);
unmentionable-expression-template-type operator*(const see-below&, const see-below&);
unmentionable-expression-template-type operator/(const see-below&, const see-below&);
// Integer only operations:
unmentionable-expression-template-type operator%(const see-below&, const see-below&);
unmentionable-expression-template-type operator&(const see-below&, const see-below&);
unmentionable-expression-template-type operator|(const see-below&, const see-below&);
unmentionable-expression-template-type operator^(const see-below&, const see-below&);
unmentionable-expression-template-type operator<<(const see-below&, const integer-type&);
unmentionable-expression-template-type operator>>(const see-below&, const integer-type&);
// Comparison operators:
bool operator==(const see-below&, const see-below&);
bool operator!=(const see-below&, const see-below&);
bool operator< (const see-below&, const see-below&);
bool operator> (const see-below&, const see-below&);
bool operator<=(const see-below&, const see-below&);
bool operator>=(const see-below&, const see-below&);
```

These operators all take their usual arithmetic meanings.

The arguments to these functions must contain at least one of the following:

- A `number`.

- An expression template type derived from `number`.

- Any type for which `number` has an implicit constructor - for example a builtin arithmetic type.

The return type of these operators is either:

- An *unmentionable-type* expression template type when `ExpressionTemplates` is `true`.

- Type `number<Backend, et_off>` when `ExpressionTemplates` is `false`.

- Type `bool` if the operator is a comparison operator.

Finally note that the second argument to the left and right shift operations must be a builtin integer type, and that the argument must be positive (negative arguments result in a `std::runtime_error` being thrown).

swap

```
template <class Backend, ExpressionTemplates>
void swap(number<Backend, ExpressionTemplates>& a, number<Backend, ExpressionTemplates>& b);
```

Swaps `a` and `b`.

Iostream Support

```
template <class Backend, expression_template_option ExpressionTemplates>
std::ostream& operator << (std::ostream& os, const number<Backend, ExpressionTemplates>& r);
template <class Unspecified...>
std::ostream& operator << (std::ostream& os, const unmentionable-expression-template& r);
template <class Backend, expression_template_option ExpressionTemplates>
inline std::istream& operator >> (std::istream& is, number<Backend, ExpressionTemplates>& r)
```

These operators provided formatted input-output operations on `number` types, and expression templates derived from them.

It's down to the back-end type to actually implement string conversion. However, the back-ends provided with this library support all of the iostream formatting flags, field width and precision settings.

Arithmetic with a higher precision result

```
template <class ResultType, class Source1 class Source2>
ResultType& add(ResultType& result, const Source1& a, const Source2& b);

template <class ResultType, class Source1 class Source2>
ResultType& subtract(ResultType& result, const Source1& a, const Source2& b);

template <class ResultType, class Source1 class Source2>
ResultType& multiply(ResultType& result, const Source1& a, const Source2& b);
```

These functions apply the named operator to the arguments *a* and *b* and store the result in *result*, returning *result*. In all cases they behave "as if" arguments *a* and *b* were first promoted to type `ResultType` before applying the operator, though particular backends may well avoid that step by way of an optimization.

The type `ResultType` must be an instance of class `number`, and the types `Source1` and `Source2` may be either instances of class `number` or native integer types. The latter is an optimization that allows arithmetic to be performed on native integer types producing an extended precision result.

Non-member standard library function support

```
unmentionable-expression-template-type   abs     (const number-or-expression-template-type&);
unmentionable-expression-template-type   fabs    (const number-or-expression-template-type&);
unmentionable-expression-template-type   sqrt    (const number-or-expression-template-type&);
unmentionable-expression-template-type   floor   (const number-or-expression-template-type&);
unmentionable-expression-template-type   ceil    (const number-or-expression-template-type&);
unmentionable-expression-template-type   trunc   (const number-or-expression-template-type&);
unmentionable-expression-template-type   itrunc  (const number-or-expression-template-type&);
unmentionable-expression-template-type   ltrunc  (const number-or-expression-template-type&);
unmentionable-expression-template-type   lltrunc (const number-or-expression-template-type&);
unmentionable-expression-template-type   round   (const number-or-expression-template-type&);
unmentionable-expression-template-type   iround  (const number-or-expression-template-type&);
unmentionable-expression-template-type   lround  (const number-or-expression-template-type&);
unmentionable-expression-template-type   llround (const number-or-expression-template-type&);
unmentionable-expression-template-type   exp     (const number-or-expression-template-type&);
unmentionable-expression-template-type   log     (const number-or-expression-template-type&);
unmentionable-expression-template-type   log10   (const number-or-expression-template-type&);
unmentionable-expression-template-type   cos     (const number-or-expression-template-type&);
unmentionable-expression-template-type   sin     (const number-or-expression-template-type&);
unmentionable-expression-template-type   tan     (const number-or-expression-template-type&);
unmentionable-expression-template-type   acos    (const number-or-expression-template-type&);
unmentionable-expression-template-type   asin    (const number-or-expression-template-type&);
unmentionable-expression-template-type   atan    (const number-or-expression-template-type&);
unmentionable-expression-template-type   cosh    (const number-or-expression-template-type&);
unmentionable-expression-template-type   sinh    (const number-or-expression-template-type&);
unmentionable-expression-template-type   tanh    (const number-or-expression-template-type&);

unmentionable-expression-template-type   ldexp (const number-or-expression-template-type&, int);
unmentionable-expression-template-type   frexp (const number-or-expression-template-type&, int*);
unmentionable-expression-template-type   pow   (const number-or-expression-template-
type&, const number-or-expression-template-type&);
unmentionable-expression-template-type   fmod  (const number-or-expression-template-
type&, const number-or-expression-template-type&);
unmentionable-expression-template-type   atan2 (const number-or-expression-template-
type&, const number-or-expression-template-type&);
```

These functions all behave exactly as their standard library C++11 counterparts do: their argument is either an instance of `number` or an expression template derived from it; If the argument is of type `number<Backend, et_off>` then that is also the return type, otherwise the return type is an expression template.

These functions are normally implemented by the Backend type. However, default versions are provided for Backend types that don't have native support for these functions. Please note however, that this default support requires the precision of the type to be a compile time constant - this means for example that the GMP MPF Backend will not work with these functions when that type is used at variable precision.

Also note that with the exception of `abs` that these functions can only be used with floating-point Backend types (if any other types such as fixed precision or complex types are added to the library later, then these functions may be extended to support those number types).

The precision of these functions is generally determined by the backend implementation. For example the precision of these functions when used with mpfr_float is determined entirely by MPFR. When these functions use our own implementations, the accuracy of the transcendental functions is generally a few epsilon. Note however, that the trigonometrical functions incur the usual accuracy loss when reducing arguments by large multiples of π. Also note that both gmp_float and cpp_dec_float have a number of guard digits beyond their stated precision, so the error rates listed for these are in some sense artificially low.

The following table shows the error rates we observe for these functions with various backend types, functions not listed here are exact (tested on Win32 with VC++10, MPFR-3.0.0, MPIR-2.1.1):

Function	mpfr_float_50	mpf_float_50	cpp_dec_float_50
sqrt	1eps	0eps	0eps
exp	1eps	0eps	0eps
log	1eps	0eps	0eps
log10	1eps	0eps	0eps
cos	700eps	0eps	0eps
sin	1eps	0eps	0eps
tan	0eps	0eps	0eps
acos	0eps	0eps	0eps
asin	0eps	0eps	0eps
atan	1eps	0eps	0eps
cosh	1045eps[1]	0eps	0eps
sinh	2eps	0eps	0eps
tanh	1eps	0eps	0eps
pow	0eps	4eps	3eps
atan2	1eps	0eps	0eps

[1] It's likely that the inherent error in the input values to our test cases are to blame here.

Traits Class Support

```
template <class T>
struct component_type;
```

If this is a type with multiple components (for example rational or complex types), then this trait has a single member `type` that is the type of those components.

```
template <class T>
struct number_category;
```

A traits class that inherits from `mpl::int_<N>` where `N` is one of the enumerated values `number_kind_integer`, `number_kind_floating_point`, `number_kind_rational`, `number_kind_fixed_point`, or `number_kind_unknown`. This traits class is specialized for any type that has `std::numeric_limits` support as well as for classes in this library: which means it can be used for generic code that must work with built in arithmetic types as well as multiprecision ones.

```
template <class T>
struct is_number;
```

A traits class that inherits from `mpl::true_` if T is an instance of `number<>`, otherwise from `mpl::false_`.

```
template <class T>
struct is_number_expression;
```

A traits class that inherits from `mpl::true_` if T is an expression template type derived from `number<>`, otherwise from `mpl::false_`.

Integer functions

In addition to functioning with types from this library, these functions are also overloaded for built in integer types if you include `<boost/multiprecision/integer.hpp>`. Further, when used with fixed precision types (whether built in integers or multiprecision ones), the functions will promote to a wider type internally when the algorithm requires it. Versions overloaded for built in integer types return that integer type rather than an expression template.

```
unmentionable-expression-template-type   pow(const number-or-expression-template-type& b, un↵
signed p);
```

Returns b^p as an expression template. Note that this function should be used with extreme care as the result can grow so large as to take "effectively forever" to compute, or else simply run the host machine out of memory. This is the one function in this category that is not overloaded for built in integer types, further, it's probably not a good idea to use it with fixed precision `cpp_int`'s either.

```
unmentionable-expression-template-type   powm(const number-or-expression-template-
type& b, const number-or-expression-template-type& p, const number-or-expression-template-type& m);
```

Returns $b^p \bmod m$ as an expression template. Fixed precision types are promoted internally to ensure accuracy.

```
template <class Backend, expression_template_option ExpressionTemplates>
void divide_qr(const number-or-expression-template-type& x, const number-or-expression-template-
type& y,
          number<Backend, ExpressionTemplates>& q, number<Backend, ExpressionTemplates>& r);
```

Divides x by y and returns both the quotient and remainder. After the call `q = x / y` and `r = x % y`.

```
template <class Integer>
Integer integer_modulus(const number-or-expression-template-type& x, Integer val);
```

Returns the absolute value of `x % val`.

```
unsigned lsb(const number-or-expression-template-type& x);
```

Returns the index of the least significant bit that is set to 1.

Throws a `std::range_error` if the argument is <= 0.

```
template <class Backend, class ExpressionTemplates>
bool bit_test(const number<Backend, ExpressionTemplates>& val, unsigned index);
```

Returns `true` if the bit at *index* in *val* is set.

```
template <class Backend, class ExpressionTemplates>
number<Backend, ExpressionTemplates>& bit_set(number<Backend, ExpressionTemplates>& val, un↵
signed index);
```

Sets the bit at *index* in *val*, and returns *val*.

```
template <class Backend, class ExpressionTemplates>
number<Backend, ExpressionTemplates>& bit_unset(number<Backend, ExpressionTemplates>& val, un↵
signed index);
```

Unsets the bit at *index* in *val*, and returns *val*.

```
template <class Backend, class ExpressionTemplates>
number<Backend, ExpressionTemplates>& bit_flip(number<Backend, ExpressionTemplates>& val, un↵
signed index);
```

Flips the bit at *index* in *val*, and returns *val*.

```
template <class Engine>
bool miller_rabin_test(const number-or-expression-template-type& n, unsigned trials, Engine& gen);
bool miller_rabin_test(const number-or-expression-template-type& n, unsigned trials);
```

Tests to see if the number *n* is probably prime - the test excludes the vast majority of composite numbers by excluding small prime factors and performing a single Fermat test. Then performs *trials* Miller-Rabin tests. Returns `false` if *n* is definitely composite, or `true` if *n* is probably prime with the probability of it being composite less than 0.25^trials. Fixed precision types are promoted internally to ensure accuracy.

Rational Number Functions

```
typename component_type<number-or-expression-template-type>::type numerator  (const number-or-
expression-template-type&);
typename component_type<number-or-expression-template-type>::type denominator(const number-or-
expression-template-type&);
```

These functions return the numerator and denominator of a rational number respectively.

Boost.Math Interoperability Support

```
namespace boost{ namespace math{

int  fpclassify      (const number-or-expression-template-type&, int);
bool isfinite        (const number-or-expression-template-type&, int);
bool isnan           (const number-or-expression-template-type&, int);
bool isinf           (const number-or-expression-template-type&, int);
bool isnormal        (const number-or-expression-template-type&, int);

}} // namespaces
```

These floating-point classification functions behave exactly as their Boost.Math equivalents.

Other Boost.Math functions and templates may also be specialized or overloaded to ensure interoperability.

std::numeric_limits support

```
namespace std{

template <class Backend, ExpressionTemplates>
struct numeric_limits<boost::multiprecision<Backend, ExpressionTemplates> >
{
   /* Usual members here */
};

}
```

Class template `std::numeric_limits` is specialized for all instantiations of `number` whose precision is known at compile time, plus those types whose precision is unlimited (though it is much less useful in those cases). It is not specialized for types whose precision can vary at compile time (such as `mpf_float`).

cpp_int

```cpp
namespace boost{ namespace multiprecision{

typedef unspecified-type limb_type;

enum cpp_integer_type    { signed_magnitude, unsigned_magnitude };
enum cpp_int_check_type  { checked, unchecked };

template <unsigned MinDigits = 0,
          unsigned MaxDits = 0,
          cpp_integer_type SignType = signed_magnitude,
          cpp_int_check_type Checked = unchecked,
          class Allocator = std::allocator<limb_type> >
class cpp_int_backend;
//
// Expression templates default to et_off if there is no allocator:
//
template <unsigned MinDigits, unsigned MaxDigits, cpp_integer_type Sign↵
Type, cpp_int_check_type Checked>
struct expression_template_default<cpp_int_backend<MinDigits, MaxDigits, SignType, Checked, void> >
{ static const expression_template_option value = et_off; };

typedef number<cpp_int_backend<> >              cpp_int;     // arbitrary precision integer
typedef rational_adaptor<cpp_int_backend<> >    cpp_rational_backend;
typedef number<cpp_rational_backend>            cpp_rational; // arbitrary precision rational ↵
number

// Fixed precision unsigned types:
typedef number<cpp_int_backend<128, 128, unsigned_magnitude, unchecked, void> >   uint128_t;
typedef number<cpp_int_backend<256, 256, unsigned_magnitude, unchecked, void> >   uint256_t;
typedef number<cpp_int_backend<512, 512, unsigned_magnitude, unchecked, void> >   uint512_t;
typedef number<cpp_int_backend<1024, 1024, unsigned_magnitude, unchecked, void> > uint1024_t;

// Fixed precision signed types:
typedef number<cpp_int_backend<128, 128, signed_magnitude, unchecked, void> >     int128_t;
typedef number<cpp_int_backend<256, 256, signed_magnitude, unchecked, void> >     int256_t;
typedef number<cpp_int_backend<512, 512, signed_magnitude, unchecked, void> >     int512_t;
typedef number<cpp_int_backend<1024, 1024, signed_magnitude, unchecked, void> >   int1024_t;

// Over again, but with checking enabled this time:
typedef number<cpp_int_backend<0, 0, signed_magnitude, checked> >          checked_cpp_int;
typedef rational_adaptor<cpp_int_backend<0, 0, signed_magnitude, checked> >     checked_cpp_ra↵
tional_backend;
typedef number<cpp_rational_backend>                                   checked_cpp_ra↵
tional;

// Checked fixed precision unsigned types:
typedef number<cpp_int_backend<128, 128, unsigned_magnitude, checked, void> >    ↵
 checked_uint128_t;
typedef number<cpp_int_backend<256, 256, unsigned_magnitude, checked, void> >    ↵
 checked_uint256_t;
typedef number<cpp_int_backend<512, 512, unsigned_magnitude, checked, void> >    ↵
 checked_uint512_t;
typedef number<cpp_int_backend<1024, 1024, unsigned_magnitude, checked, void> > ↵
 checked_uint1024_t;

// Fixed precision signed types:
```

```
typedef number<cpp_int_backend<128, 128, signed_magnitude, checked, void> >      checked_int128_t;
typedef number<cpp_int_backend<256, 256, signed_magnitude, checked, void> >      checked_int256_t;
typedef number<cpp_int_backend<512, 512, signed_magnitude, checked, void> >      checked_int512_t;
typedef number<cpp_int_backend<1024, 1024, signed_magnitude, checked, void> >   ↵
 checked_int1024_t;

}} // namespaces
```

Class template `cpp_int_backend` fulfils all of the requirements for a Backend type. Its members and non-member functions are deliberately not documented: these are considered implementation details that are subject to change.

The template arguments are:

MinBits   Determines the number of Bits to store directly within the object before resorting to dynamic memory allocation. When zero, this field is determined automatically based on how many bits can be stored in union with the dynamic storage header: setting a larger value may improve performance as larger integer values will be stored internally before memory allocation is required.

MaxBits   Determines the maximum number of bits to be stored in the type: resulting in a fixed precision type. When this value is the same as MinBits, then the Allocator parameter is ignored, as no dynamic memory allocation will ever be performed: in this situation the Allocator parameter should be set to type `void`. Note that this parameter should not be used simply to prevent large memory allocations, not only is that role better performed by the allocator, but fixed precision integers have a tendency to allocate all of MaxBits of storage more often than one would expect.

SignType   Determines whether the resulting type is signed or not. Note that for arbitrary precision types this parameter must be `signed_magnitude`. For fixed precision types then this type may be either `signed_magnitude` or `unsigned_magnitude`.

Checked   This parameter has two values: `checked` or `unchecked`. See the tutorial for more information.

Allocator   The allocator to use for dynamic memory allocation, or type `void` if MaxBits == MinBits.

The type of `number_category<cpp_int<Args...> >::type` is `mpl::int_<number_kind_integer>`.

More information on this type can be found in the tutorial.

gmp_int

```
namespace boost{ namespace multiprecision{

class gmp_int;

typedef number<gmp_int >         mpz_int;

}} // namespaces
```

Class template `gmp_int` fulfils all of the requirements for a Backend type. Its members and non-member functions are deliberately not documented: these are considered implementation details that are subject to change.

The type of `number_category<cpp_int<Args...> >::type` is `mpl::int_<number_kind_integer>`.

More information on this type can be found in the tutorial.

tom_int

```
namespace boost{ namespace multiprecision{

class tommath_int;

typedef number<tommath_int >        tom_int;

}} // namespaces
```

Class template `tommath_int` fulfils all of the requirements for a [Backend](#) type. Its members and non-member functions are deliberately not documented: these are considered implementation details that are subject to change.

The type of `number_category<cpp_int<Args...> >::type` is `mpl::int_<number_kind_integer>`.

More information on this type can be found in the [tutorial](#).

gmp_float

```
namespace boost{ namespace multiprecision{

template <unsigned Digits10>
class gmp_float;

typedef number<gmp_float<50> >    mpf_float_50;
typedef number<gmp_float<100> >   mpf_float_100;
typedef number<gmp_float<500> >   mpf_float_500;
typedef number<gmp_float<1000> >  mpf_float_1000;
typedef number<gmp_float<0> >     mpf_float;

}} // namespaces
```

Class template `gmp_float` fulfils all of the requirements for a [Backend](#) type. Its members and non-member functions are deliberately not documented: these are considered implementation details that are subject to change.

The class takes a single template parameter - `Digits10` - which is the number of decimal digits precision the type should support. When this parameter is zero, then the precision can be set at runtime via `number::default_precision` and `number::precision`. Note that this type does not in any way change the GMP library's global state (for example it does not change the default precision of the mpf_t data type), therefore you can safely mix this type with existing code that uses GMP, and also mix `gmp_floats` of differing precision.

The type of `number_category<cpp_int<Args...> >::type` is `mpl::int_<number_kind_floating_point>`.

More information on this type can be found in the [tutorial](#).

mpfr_float_backend

```
namespace boost{ namespace multiprecision{

template <unsigned Digits10>
class mpfr_float_backend;

typedef number<mpfr_float_backend<50> >    mpfr_float_50;
typedef number<mpfr_float_backend<100> >   mpfr_float_100;
typedef number<mpfr_float_backend<500> >   mpfr_float_500;
typedef number<mpfr_float_backend<1000> >  mpfr_float_1000;
typedef number<mpfr_float_backend<0> >     mpfr_float;

}} // namespaces
```

Class template `mpfr_float_backend` fulfils all of the requirements for a Backend type. Its members and non-member functions are deliberately not documented: these are considered implementation details that are subject to change.

The class takes a single template parameter - `Digits10` - which is the number of decimal digits precision the type should support. When this parameter is zero, then the precision can be set at runtime via `number::default_precision` and `number::precision`. Note that this type does not in any way change the GMP or MPFR library's global state (for example it does not change the default precision of the mpfr_t data type), therefore you can safely mix this type with existing code that uses GMP or MPFR, and also mix `mpfr_float_backend`s of differing precision.

The type of `number_category<cpp_int<Args...> >::type` is `mpl::int_<number_kind_floating_point>`.

More information on this type can be found in the tutorial.

cpp_dec_float

```
namespace boost{ namespace multiprecision{

template <unsigned Digits10, class ExponentType = boost::int32_t, class Allocator = void>
class cpp_dec_float;

typedef number<cpp_dec_float<50> > cpp_dec_float_50;
typedef number<cpp_dec_float<100> > cpp_dec_float_100;

}} // namespaces
```

Class template `cpp_dec_float` fulfils all of the requirements for a Backend type. Its members and non-member functions are deliberately not documented: these are considered implementation details that are subject to change.

The class takes three template parameters:

Digits10	The number of decimal digits precision the type should support. Note that this type does not normally perform any dynamic memory allocation, and as a result the `Digits10` template argument should not be set too high or the class's size will grow unreasonably large.
ExponentType	A signed integer type that represents the exponent of the number
Allocator	The allocator used: defaults to type `void`, meaning all storage is within the class, and no dynamic allocation is performed, but can be set to a standard library allocator if dynamic allocation makes more sense.

The type of `number_category<cpp_int<Args...> >::type` is `mpl::int_<number_kind_floating_point>`.

More information on this type can be found in the tutorial.

Internal Support Code

There are some traits classes which authors of new backends should be aware of:

```
namespace boost{ namespace multiprecision{ namespace detail{

template<typename From, typename To>
struct is_explicitly_convertible;

}}}
```

Inherits from `boost::integral_constant<bool,true>` if type `From` has an explicit conversion from `To`.

For compilers that support C++11 SFINAE-expressions this trait should "just work". Otherwise it inherits from `boost::is_convertible<From, To>::type`, and will need to be specialised for Backends that have constructors marked as `explicit`.

```
template <class From, class To>
struct is_lossy_conversion
{
   static const bool value = see below;
};
```

Member `value` is true if the conversion from `From` to `To` would result in a loss of precision, and `false` otherwise.

The default version of this trait simply checks whether the *kind* of conversion (for example from a floating point to an integer type) is inherently lossy. Note that if either of the types `From` or `To` are of an unknown number category (because `number_category` is not specialised for that type) then this trait will be `true`.

```
template<typename From, typename To>
struct is_restricted_conversion
{
   static const bool value = see below;
};
```

Member `value` is `true` if `From` is only explicitly convertible to `To` and not implicitly convertible, or if `is_lossy_conversion<From, To>::value` is `true`. Otherwise `false`.

Note that while this trait is the ultimate arbiter of which constructors are marked as `explicit` in class `number`, authors of backend types should generally specialise one of the traits above, rather than this one directly.

```
template <class T>
is_signed_number;
template <class T>
is_unsigned_number;
```

These two traits inherit from either `mpl::true_` or `mpl::false_`, by default types are assumed to be signed unless `is_unsigned_number` is specialized for that type.

Backend Requirements

The requirements on the `Backend` template argument to `number` are split up into sections: compulsory and optional.

Compulsory requirements have no default implementation in the library, therefore if the feature they implement is to be supported at all, then they must be implemented by the backend.

Optional requirements have default implementations that are called if the backend doesn't provide it's own. Typically the backend will implement these to improve performance.

In the following tables, type B is the `Backend` template argument to `number`, `b` and `b2` are a variables of type B, `cb`, `cb2` and `cb3` are constant variables of type `const B`, `rb` is a variable of type `B&&`, `a` and `a2` are variables of Arithmetic type, `s` is a variable of type `const char*`, `ui` is a variable of type `unsigned`, `bb` is a variable of type `bool`, `pa` is a variable of type pointer-to-arithmetic-type, `exp` is a variable of type `B::exp_type`, `pexp` is a variable of type `B::exp_type*`, `i` is a variable of type `int`, `pi` pointer to a variable of type `int`, B2 is another type that meets these requirements, `b2` is a variable of type B2, `ss` is variable of type `std::streamsize` and `ff` is a variable of type `std::ios_base::fmtflags`.

Table 4. Compulsory Requirements on the Backend type.

Expression	Return Type	Comments	Throws
`B::signed_types`	`mpl::list<type-list>`	A list of signed integral types that can be assigned to type B. The types shall be listed in order of size, smallest first, and shall terminate in the type that is `std::intmax_t`.	
`B::unsigned_types`	`mpl::list<type-list>`	A list of unsigned integral types that can be assigned to type B. The types shall be listed in order of size, smallest first, and shall terminate in the type that is `std::uintmax_t`.	
`B::float_types`	`mpl::list<type-list>`	A list of floating-point types that can be assigned to type B.The types shall be listed in order of size, smallest first, and shall terminate in type `long double`.	
`B::exponent_type`	A signed integral type.	The type of the exponent of type B. This type is required only for floating point types.	
`B()`		Default constructor.	
`B(cb)`		Copy Constructor.	
`b = b`	`B&`	Assignment operator.	
`b = a`	`B&`	Assignment from an Arithmetic type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`.	
`b = s`	`B&`	Assignment from a string.	Throws a `std::runtime_error` if the string could not be interpreted as a valid number.
`b.swap(b)`	`void`	Swaps the contents of its arguments.	`noexcept`
`cb.str(ss, ff)`	`std::string`	Returns the string representation of `b` with `ss` digits and formatted according to the flags set in `ff`. If `ss` is zero, then returns as many digits as are required to reconstruct the original value.	

Expression	Return Type	Comments	Throws
`b.negate()`	`void`	Negates `b`.	
`cb.compare(cb2)`	`int`	Compares `cb` and `cb2`, returns a value less than zero if `cb` < `cb2`, a value greater than zero if `cb` > `cb2` and zero if `cb` == `cb2`.	`noexcept`
`cb.compare(a)`	`int`	Compares `cb` and `a`, returns a value less than zero if `cb` < `a`, a value greater than zero if `cb` > `a` and zero if `cb` == `a`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`.	
`eval_add(b, cb)`	`void`	Adds `cb` to `b`.	
`eval_subtract(b, cb)`	`void`	Subtracts `cb` from `b`.	
`eval_multiply(b, cb)`	`void`	Multiplies `b` by `cb`.	
`eval_divide(b, cb)`	`void`	Divides `b` by `cb`.	`std::overflow_error` if `cb` has the value zero, and `std::numeric_limits<number<B> >::has_infinity == false`
`eval_modulus(b, cb)`	`void`	Computes `b %= cb`, only required when `B` is an integer type.	`std::overflow_error` if `cb` has the value zero.
`eval_bitwise_and(b, cb)`	`void`	Computes `b &= cb`, only required when `B` is an integer type.	
`eval_bitwise_or(b, cb)`	`void`	Computes `b |= cb`, only required when `B` is an integer type.	
`eval_bitwise_xor(b, cb)`	`void`	Computes `b ^= cb`, only required when `B` is an integer type.	
`eval_complement(b, cb)`	`void`	Computes the ones-complement of `cb` and stores the result in `b`, only required when `B` is an integer type.	
`eval_left_shift(b, ui)`	`void`	Computes `b <<= ui`, only required when `B` is an integer type.	

Expression	Return Type	Comments	Throws
`eval_right_shift(b, ui)`	`void`	Computes `b >>= ui`, only required when `B` is an integer type.	
`eval_convert_to(pa, cb)`	`void`	Converts `cb` to the type of `*pa` and store the result in `*pa`. Type `B` shall support conversion to at least types `std::intmax_t`, `std::uintmax_t` and `long long`. Conversion to other arithmetic types can then be synthesised using other operations. Conversions to other types are entirely optional.	
`eval_frexp(b, cb, pexp)`	`void`	Stores values in `b` and `*pexp` such that the value of `cb` is $b * 2^{*pexp}$, only required when `B` is a floating-point type.	
`eval_ldexp(b, cb, exp)`	`void`	Stores a value in `b` that is $cb * 2^{exp}$, only required when `B` is a floating-point type.	
`eval_frexp(b, cb, pi)`	`void`	Stores values in `b` and `*pi` such that the value of `cb` is $b * 2^{*pi}$, only required when `B` is a floating-point type.	`std::runtime_error` if the exponent of `cb` is too large to be stored in an `int`.
`eval_ldexp(b, cb, i)`	`void`	Stores a value in `b` that is $cb * 2^{i}$, only required when `B` is a floating-point type.	
`eval_floor(b, cb)`	`void`	Stores the floor of `cb` in `b`, only required when `B` is a floating-point type.	
`eval_ceil(b, cb)`	`void`	Stores the ceiling of `cb` in `b`, only required when `B` is a floating-point type.	
`eval_sqrt(b, cb)`	`void`	Stores the square root of `cb` in `b`, only required when `B` is a floating-point type.	

Expression	Return Type	Comments	Throws
`boost::multiprecision::number_category<B>::type`	`mpl::int_<N>`	`N` is one of the values `number_kind_integer`, `number_kind_floating_point`, `number_kind_rational` or `number_kind_fixed_point`. Defaults to `number_kind_floating_point`.	

Table 5. Optional Requirements on the Backend Type

Expression	Returns	Comments	Throws
Construct and assign:			
`B(rb)`	B	Move constructor. Afterwards variable `rb` shall be in sane state, albeit with unspecified value. Only destruction and assignment to the moved-from variable `rb` need be supported after the operation.	`noexcept`
`b = rb`	B&	Move-assign. Afterwards variable `rb` shall be in sane state, albeit with unspecified value. Only destruction and assignment to the moved-from variable `rb` need be supported after the operation.	`noexcept`
`B(a)`	B	Direct construction from an arithmetic type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, this operation is simulated using default-construction followed by assignment.	
`B(b2)`	B	Copy constructor from a different back-end type. When not provided, a generic interconversion routine is used. This constructor may be `explicit` if the corresponding frontend constructor should also be `explicit`.	
`b = b2`	b&	Assignment operator from a different back-end type. When not provided, a generic interconversion routine is used.	
`assign_components(b, a, a)`	void	Assigns to `b` the two components in the following arguments. Only applies to rational and complex number types. When not provided, arithmetic operations are used to synthesise the result from the two values.	

Expression	Returns	Comments	Throws
`assign_components(b, b2, b2)`	void	Assigns to `b` the two components in the following arguments. Only applies to rational and complex number types. When not provided, arithmetic operations are used to synthesise the result from the two values.	
Comparisons:			
`eval_eq(cb, cb2)`	bool	Returns `true` if `cb` and `cb2` are equal in value. When not provided, the default implementation returns `cb.compare(cb2) == 0`.	noexcept
`eval_eq(cb, a)`	bool	Returns `true` if `cb` and `a` are equal in value. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, return the equivalent of `eval_eq(cb, B(a))`.	
`eval_eq(a, cb)`	bool	Returns `true` if `cb` and `a` are equal in value. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, the default version returns `eval_eq(cb, a)`.	
`eval_lt(cb, cb2)`	bool	Returns `true` if `cb` is less than `cb2` in value. When not provided, the default implementation returns `cb.compare(cb2) < 0`.	noexcept
`eval_lt(cb, a)`	bool	Returns `true` if `cb` is less than `a` in value. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, the default implementation returns `eval_lt(cb, B(a))`.	

Expression	Returns	Comments	Throws
`eval_lt(a, cb)`	`bool`	Returns `true` if `a` is less than `cb` in value. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, the default implementation returns `eval_gt(cb, a)`.	
`eval_gt(cb, cb2)`	`bool`	Returns `true` if `cb` is greater than `cb2` in value. When not provided, the default implementation returns `cb.compare(cb2) > 0`.	`noexcept`
`eval_gt(cb, a)`	`bool`	Returns `true` if `cb` is greater than `a` in value. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, the default implementation returns `eval_gt(cb, B(a))`.	
`eval_gt(a, cb)`	`bool`	Returns `true` if `a` is greater than `cb` in value. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, the default implementation returns `eval_lt(cb, a)`.	
`eval_is_zero(cb)`	`bool`	Returns `true` if `cb` is zero, otherwise `false`. The default version of this function returns `cb.compare(ui_type(0)) == 0`, where `ui_type` is `ui_type` is `typename mpl::front<typename B::unsigned_types>::type`.	

Expression	Returns	Comments	Throws
`eval_get_sign(cb)`	`int`	Returns a value < zero if `cb` is negative, a value > zero if `cb` is positive, and zero if `cb` is zero. The default version of this function returns `cb.compare(ui_type(0))`, where `ui_type` is `ui_type` is `typename mpl::front<typename B::unsigned_types>::type`.	
Basic arithmetic:			
`eval_add(b, a)`	`void`	Adds `a` to `b`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, the default version calls `eval_add(b, B(a))`	
`eval_add(b, cb, cb2)`	`void`	Add `cb` to `cb2` and stores the result in `b`. When not provided, does the equivalent of `b = cb; eval_add(b, cb2)`.	
`eval_add(b, cb, a)`	`void`	Add `cb` to `a` and stores the result in `b`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_add(b, cb, B(a))`.	
`eval_add(b, a, cb)`	`void`	Add `a` to `cb` and stores the result in `b`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_add(b, cb, a)`.	
`eval_subtract(b, a)`	`void`	Subtracts `a` from `b`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, the default version calls `eval_subtract(b, B(a))`	

Expression	Returns	Comments	Throws
`eval_subtract(b, cb, cb2)`	`void`	Subtracts `cb2` from `cb` and stores the result in `b`. When not provided, does the equivalent of `b = cb; eval_subtract(b, cb2)`.	
`eval_subtract(b, cb, a)`	`void`	Subtracts `a` from `cb` and stores the result in `b`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_subtract(b, cb, B(a))`.	
`eval_subtract(b, a, cb)`	`void`	Subtracts `cb` from `a` and stores the result in `b`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_subtract(b, cb, a); b.negate();`.	
`eval_multiply(b, a)`	`void`	Multiplies `b` by `a`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, the default version calls `eval_multiply(b, B(a))`	
`eval_multiply(b, cb, cb2)`	`void`	Multiplies `cb` by `cb2` and stores the result in `b`. When not provided, does the equivalent of `b = cb; eval_multiply(b, cb2)`.	
`eval_multiply(b, cb, a)`	`void`	Multiplies `cb` by `a` and stores the result in `b`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_multiply(b, cb, B(a))`.	

Expression	Returns	Comments	Throws
`eval_multiply(b, a, cb)`	`void`	Multiplies `a` by `cb` and stores the result in `b`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_multiply(b, cb, a)`.	
`eval_multiply_add(b, cb, cb2)`	`void`	Multiplies `cb` by `cb2` and adds the result to `b`. When not provided does the equivalent of creating a temporary `B t` and `eval_multiply(t, cb, cb2)` followed by `eval_add(b, t)`.	
`eval_multiply_add(b, cb, a)`	`void`	Multiplies `a` by `cb` and adds the result to `b`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided does the equivalent of creating a temporary `B t` and `eval_multiply(t, cb, a)` followed by `eval_add(b, t)`.	
`eval_multiply_add(b, a, cb)`	`void`	Multiplies `a` by `cb` and adds the result to `b`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided does the equivalent of `eval_multiply_add(b, cb, a)`.	
`eval_multiply_subtract(b, cb, cb2)`	`void`	Multiplies `cb` by `cb2` and subtracts the result from `b`. When not provided does the equivalent of creating a temporary `B t` and `eval_multiply(t, cb, cb2)` followed by `eval_subtract(b, t)`.	

Expression	Returns	Comments	Throws
`eval_multiply_subtract(b, cb, a)`	`void`	Multiplies `a` by `cb` and subtracts the result from `b`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided does the equivalent of creating a temporary `B t` and `eval_multiply(t, cb, a)` followed by `eval_subtract(b, t)`.	
`eval_multiply_subtract(b, a, cb)`	`void`	Multiplies `a` by `cb` and subtracts the result from `b`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided does the equivalent of `eval_multiply_subtract(b, cb, a)`.	
`eval_multiply_add(b, cb, cb2, cb3)`	`void`	Multiplies `cb` by `cb2` and adds the result to `cb3` storing the result in `b`. When not provided does the equivalent of `eval_multiply(b, cb, cb2)` followed by `eval_add(b, cb3)`. For brevity, only a version showing all arguments of type `B` is shown here, but you can replace up to any 2 of `cb`, `cb2` and `cb3` with any type listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`.	
`eval_multiply_subtract(b, cb, cb2, cb3)`	`void`	Multiplies `cb` by `cb2` and subtracts from the result `cb3` storing the result in `b`. When not provided does the equivalent of `eval_multiply(b, cb, cb2)` followed by `eval_subtract(b, cb3)`. For brevity, only a version showing all arguments of type `B` is shown here, but you can replace up to any 2 of `cb`, `cb2` and `cb3` with any type listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`.	

Expression	Returns	Comments	Throws
`eval_divide(b, a)`	`void`	Divides `b` by `a`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, the default version calls `eval_divide(b, B(a))`	`std::overflow_error` if `a` has the value zero, and `std::numeric_limits<number<B> >::has_infinity == false`
`eval_divide(b, cb, cb2)`	`void`	Divides `cb` by `cb2` and stores the result in `b`. When not provided, does the equivalent of `b = cb; eval_divide(b, cb2)`.	`std::overflow_error` if `cb2` has the value zero, and `std::numeric_limits<number<B> >::has_infinity == false`
`eval_divide(b, cb, a)`	`void`	Divides `cb` by `a` and stores the result in `b`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_divide(b, cb, B(a))`.	`std::overflow_error` if `a` has the value zero, and `std::numeric_limits<number<B> >::has_infinity == false`
`eval_divide(b, a, cb)`	`void`	Divides `a` by `cb` and stores the result in `b`. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_divide(b, B(a), cb)`.	`std::overflow_error` if `cb` has the value zero, and `std::numeric_limits<number<B> >::has_infinity == false`
`eval_increment(b)`	`void`	Increments the value of `b` by one. When not provided, does the equivalent of `eval_add(b, static_cast<ui_type>(1u))`. Where `ui_type` is `typename mpl::front<typename B::unsigned_types>::type`.	
`eval_decrement(b)`	`void`	Decrements the value of `b` by one. When not provided, does the equivalent of `eval_subtract(b, static_cast<ui_type>(1u))`. Where `ui_type` is `typename mpl::front<typename B::unsigned_types>::type`.	

Expression	Returns	Comments	Throws
Integer specific operations:			
`eval_modulus(b, a)`	`void`	Computes `b %= cb`, only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, the default version calls `eval_modulus(b, B(a))`	`std::overflow_error` if `a` has the value zero.
`eval_modulus(b, cb, cb2)`	`void`	Computes `cb % cb2` and stores the result in `b`, only required when `B` is an integer type. When not provided, does the equivalent of `b = cb; eval_modulus(b, cb2)`.	`std::overflow_error` if `a` has the value zero.
`eval_modulus(b, cb, a)`	`void`	Computes `cb % a` and stores the result in `b`, only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_modulus(b, cb, B(a))`.	`std::overflow_error` if `a` has the value zero.
`eval_modulus(b, a, cb)`	`void`	Computes `cb % a` and stores the result in `b`, only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_modulus(b, B(a), cb)`.	`std::overflow_error` if `a` has the value zero.
`eval_bitwise_and(b, a)`	`void`	Computes `b &= cb`, only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, the default version calls `eval_bitwise_and(b, B(a))`	

Expression	Returns	Comments	Throws
`eval_bitwise_and(b, cb, cb2)`	`void`	Computes `cb & cb2` and stores the result in `b`, only required when `B` is an integer type. When not provided, does the equivalent of `b = cb; eval_bitwise_and(b, cb2)`.	
`eval_bitwise_and(b, cb, a)`	`void`	Computes `cb & a` and stores the result in `b`, only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_bitwise_and(b, cb, B(a))`.	
`eval_bitwise_and(b, a, cb)`	`void`	Computes `cb & a` and stores the result in `b`, only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_bitwise_and(b, cb, a)`.	
`eval_bitwise_or(b, a)`	`void`	Computes `b |= cb`, only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, the default version calls `eval_bitwise_or(b, B(a))`	
`eval_bitwise_or(b, cb, cb2)`	`void`	Computes `cb	cb2` and stores the result in `b`, only required when `B` is an integer type. When not provided, does the equivalent of `b = cb; eval_bitwise_or(b, cb2)`.	

Expression	Returns	Comments	Throws
`eval_bitwise_or(b, cb, a)`	`void`	Computes `cb	a` and stores the result in `b`, only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_bitwise_or(b, cb, B(a))`.	
`eval_bitwise_or(b, a, cb)`	`void`	Computes `cb	a` and stores the result in `b`, only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_bitwise_or(b, cb, a)`.	
`eval_bitwise_xor(b, a)`	`void`	Computes `b ^= cb`, only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, the default version calls `eval_bitwise_xor(b, B(a))`	
`eval_bitwise_xor(b, cb, cb2)`	`void`	Computes `cb ^ cb2` and stores the result in `b`, only required when `B` is an integer type. When not provided, does the equivalent of `b = cb; eval_bitwise_xor(b, cb2)`.	
`eval_bitwise_xor(b, cb, a)`	`void`	Computes `cb ^ a` and stores the result in `b`, only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_bitwise_xor(b, cb, B(a))`.	

Expression	Returns	Comments	Throws
`eval_bitwise_xor(b, a, cb)`	`void`	Computes `a ^ cb` and stores the result in `b`, only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. When not provided, does the equivalent of `eval_bitwise_xor(b, cb, a)`.	
`eval_left_shift(b, cb, ui)`	`void`	Computes `cb << ui` and stores the result in `b`, only required when `B` is an integer type. When not provided, does the equivalent of `b = cb; eval_left_shift(b, a);`.	
`eval_right_shift(b, cb, ui)`	`void`	Computes `cb >> ui` and stores the result in `b`, only required when `B` is an integer type. When not provided, does the equivalent of `b = cb; eval_right_shift(b, a);`.	
`eval_qr(cb, cb2, b, b2)`	`void`	Sets `b` to the result of `cb / cb2` and `b2` to the result of `cb % cb2`. Only required when `B` is an integer type. The default version of this function is synthesised from other operations above.	`std::overflow_error` if a has the value zero.
`eval_integer_modulus(cb, ui)`	`unsigned`	Returns the result of `cb % ui`. Only required when `B` is an integer type. The default version of this function is synthesised from other operations above.	`std::overflow_error` if a has the value zero.
`eval_lsb(cb)`	`unsigned`	Returns the index of the least significant bit that is set. Only required when `B` is an integer type. The default version of this function is synthesised from other operations above.	
`eval_bit_test(cb, ui)`	`bool`	Returns true if `cb` has bit `ui` set. Only required when `B` is an integer type. The default version of this function is synthesised from other operations above.	

Expression	Returns	Comments	Throws
`eval_bit_set(b, ui)`	`void`	Sets the bit at index `ui` in `b`. Only required when `B` is an integer type. The default version of this function is synthesised from other operations above.	
`eval_bit_unset(b, ui)`	`void`	Unsets the bit at index `ui` in `b`. Only required when `B` is an integer type. The default version of this function is synthesised from other operations above.	
`eval_bit_flip(b, ui)`	`void`	Flips the bit at index `ui` in `b`. Only required when `B` is an integer type. The default version of this function is synthesised from other operations above.	
`eval_gcd(b, cb, cb2)`	`void`	Sets `b` to the greatest common divisor of `cb` and `cb2`. Only required when `B` is an integer type. The default version of this function is synthesised from other operations above.	
`eval_lcm(b, cb, cb2)`	`void`	Sets `b` to the least common multiple of `cb` and `cb2`. Only required when `B` is an integer type. The default version of this function is synthesised from other operations above.	
`eval_gcd(b, cb, a)`	`void`	Sets `b` to the greatest common divisor of `cb` and `cb2`. Only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. The default version of this function calls `eval_gcd(b, cb, B(a))`.	

Expression	Returns	Comments	Throws
`eval_lcm(b, cb, a)`	`void`	Sets `b` to the least common multiple of `cb` and `cb2`. Only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. The default version of this function calls `eval_lcm(b, cb, B(a))`.	
`eval_gcd(b, a, cb)`	`void`	Sets `b` to the greatest common divisor of `cb` and `a`. Only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. The default version of this function calls `eval_gcd(b, cb, a)`.	
`eval_lcm(b, a, cb)`	`void`	Sets `b` to the least common multiple of `cb` and `a`. Only required when `B` is an integer type. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types` or `B::float_types`. The default version of this function calls `eval_lcm(b, cb, a)`.	
`eval_powm(b, cb, cb2, cb3)`	`void`	Sets `b` to the result of *(cb^cb2)%cb3*. The default version of this function is synthesised from other operations above.	
`eval_powm(b, cb, cb2, a)`	`void`	Sets `b` to the result of *(cb^cb2)%a*. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types`. The default version of this function is synthesised from other operations above.	

Expression	Returns	Comments	Throws
`eval_powm(b, cb, a, cb2)`	`void`	Sets `b` to the result of *(cb^a)%cb2*. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types`. The default version of this function is synthesised from other operations above.	
`eval_powm(b, cb, a, a2)`	`void`	Sets `b` to the result of *(cb^a)%a2*. The type of `a` shall be listed in one of the type lists `B::signed_types`, `B::unsigned_types`. The default version of this function is synthesised from other operations above.	
Sign manipulation:			
`eval_abs(b, cb)`	`void`	Set `b` to the absolute value of `cb`. The default version of this functions assigns `cb` to `b`, and then calls `b.negate()` if `eval_get_sign(cb) < 0`.	
`eval_fabs(b, cb)`	`void`	Set `b` to the absolute value of `cb`. The default version of this functions assigns `cb` to `b`, and then calls `b.negate()` if `eval_get_sign(cb) < 0`.	
Floating point functions:			
`eval_fpclassify(cb)`	`int`	Returns one of the same values returned by `std::fpclassify`. Only required when `B` is an floating-point type. The default version of this function will only test for zero `cb`.	
`eval_trunc(b, cb)`	`void`	Performs the equivalent operation to `std::trunc` on argument `cb` and stores the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	

Expression	Returns	Comments	Throws
`eval_round(b, cb)`	`void`	Performs the equivalent operation to `std::round` on argument `cb` and stores the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	
`eval_exp(b, cb)`	`void`	Performs the equivalent operation to `std::exp` on argument `cb` and stores the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	
`eval_log(b, cb)`	`void`	Performs the equivalent operation to `std::log` on argument `cb` and stores the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	
`eval_log10(b, cb)`	`void`	Performs the equivalent operation to `std::log10` on argument `cb` and stores the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	
`eval_sin(b, cb)`	`void`	Performs the equivalent operation to `std::sin` on argument `cb` and stores the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	
`eval_cos(b, cb)`	`void`	Performs the equivalent operation to `std::cos` on argument `cb` and stores the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	

Expression	Returns	Comments	Throws
`eval_tan(b, cb)`	`void`	Performs the equivalent operation to `std::exp` on argument `cb` and stores the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	
`eval_asin(b, cb)`	`void`	Performs the equivalent operation to `std::asin` on argument `cb` and stores the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	
`eval_acos(b, cb)`	`void`	Performs the equivalent operation to `std::acos` on argument `cb` and stores the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	
`eval_atan(b, cb)`	`void`	Performs the equivalent operation to `std::atan` on argument `cb` and stores the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	
`eval_sinh(b, cb)`	`void`	Performs the equivalent operation to `std::sinh` on argument `cb` and stores the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	
`eval_cosh(b, cb)`	`void`	Performs the equivalent operation to `std::cosh` on argument `cb` and stores the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	

Expression	Returns	Comments	Throws
`eval_tanh(b, cb)`	*void*	Performs the equivalent operation to `std::tanh` on argument `cb` and stores the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	
`eval_fmod(b, cb, cb2)`	*void*	Performs the equivalent operation to `std::fmod` on arguments `cb` and `cb2`, and store the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	
`eval_pow(b, cb, cb2)`	*void*	Performs the equivalent operation to `std::pow` on arguments `cb` and `cb2`, and store the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	
`eval_atan2(b, cb, cb2)`	*void*	Performs the equivalent operation to `std::atan` on arguments `cb` and `cb2`, and store the result in `b`. Only required when `B` is an floating-point type. The default version of this function is synthesised from other operations above.	

When the tables above place no *throws* requirements on an operation, then it is up to each type modelling this concept to decide when or whether throwing an exception is desirable. However, thrown exceptions should always either be the type, or inherit from the type `std::runtime_error`. For example, a floating point type might choose to throw `std::overflow_error` whenever the result of an operation would be infinite, and `std::underflow_error` whenever it would round to zero.

> **Note**
>
> The non-member functions are all named with an "eval_" prefix to avoid conflicts with template classes of the same name - in point of fact this naming convention shouldn't be necessary, but rather works around some compiler bugs.

Header File Structure

Table 6. Top level headers

Header	Contains
cpp_int.hpp	The `cpp_int` backend type.
gmp.hpp	Defines all GMP related backends.
miller_rabin.hpp	Miller Rabin primality testing code.
number.hpp	Defines the `number` backend, is included by all the backend headers.
mpfr.hpp	Defines the mpfr_float_backend backend.
random.hpp	Defines code to interoperate with Boost.Random.
rational_adaptor.hpp	Defines the `rational_adaptor` backend.
cpp_dec_float.hpp	Defines the `cpp_dec_float` backend.
tommath.hpp	Defines the `tommath_int` backend.
concepts/number_archetypes.hpp	Defines a backend concept archetypes for testing use.

Table 7. Implementation Headers]

Header	Contains
cpp_int/add.hpp	Add and subtract operators for `cpp_int_backend`.
cpp_int/bitwise.hpp	Bitwise operators for `cpp_int_backend`.
cpp_int/checked.hpp	Helper functions for checked arithmetic for `cpp_int_backend`.
cpp_int/comparison.hpp	Comparison operators for `cpp_int_backend`.
cpp_int/cpp_int_config.hpp	Basic setup and configuration for `cpp_int_backend`.
cpp_int/divide.hpp	Division and modulus operators for `cpp_int_backend`.
cpp_int/limits.hpp	`numeric_limits` support for `cpp_int_backend`.
cpp_int/misc.hpp	Miscellaneous operators for `cpp_int_backend`.
cpp_int/multiply.hpp	Multiply operators for `cpp_int_backend`.
detail/big_lanczos.hpp	Lanczos support for Boost.Math integration.
detail/default_ops.hpp	Default versions of the optional backend non-member functions.
detail/generic_interconvert.hpp	Generic interconversion routines.
detail/number_base.hpp	All the expression template code, metaprogramming, and operator overloads for `number`.
detail/no_et_ops.hpp	The non-expression template operators.
defail/functions/constants.hpp	Defines constants used by the floating point functions.
detail/functions/pow.hpp	Defines default versions of the power and exponential related floating point functions.
detail/functions/trig.hpp	Defines default versions of the trigonometric related floating point functions.

Performance Comparison

The Overhead in the Number Class Wrapper

Using a simple backend class that wraps any built in arithmetic type we can measure the overhead involved in wrapping a type inside the `number` frontend, and the effect that turning on expression templates has. The following table compares the performance between `double` and a `double` wrapped inside class `number`:

Type	Bessel Function Evaluation	Non-Central T Evaluation
`double`	**1.0 (0.016s)**	**1.0** (0.46s)
`number<arithmetic_backend<double>, et_off>`	1.2 (0.019s)	**1.0**(0.46s)
`number<arithmetic_backend<double>, et_on>`	1.2 (0.019s)	1.7 (0.79s)

As you can see whether or not there is an overhead, and how large it is depends on the actual situation, but the overhead is in any cases small. Expression templates generally add a greater overhead the more complex the expression becomes due to the logic of figuring out how to best unpack and evaluate the expression, but of course this is also the situation where you save more temporaries. For a "trivial" backend like this, saving temporaries has no benefit, but for larger types it becomes a bigger win.

The following table compares arithmetic using either `long long` or `number<arithmetic_backend<long long> >` for the voronoi-diagram builder test:

Type	Relative time
`long long`	**1.0**(0.0823s)
`number<arithmetic_backend<long long>, et_off>`	1.05 (0.0875s)

This test involves mainly creating a lot of temporaries and performing a small amount of arithmetic on them, with very little difference in performance between the native and "wrapped" types.

The test code was compiled with Microsoft Visual Studio 2010 with all optimisations turned on (/Ox), and used MPIR-2.3.0 and libtommath-0.42.0. The tests were run on 32-bit Windows Vista machine.

Floating Point Real World Tests

These tests test the total time taken to execute all of Boost.Math's test cases for these functions. In each case the best performing library gets a relative score of 1, with the total execution time given in brackets. The first three libraries listed are the various floating point types provided by this library, while for comparison, two popular C++ front-ends to MPFR (mpfr_class and mpreal) are also shown.

Table 8. Bessel Function Performance

Library	50 Decimal Digits	100 Decimal Digits
mpfr_float	1.2 (5.78s)	1.2 (9.56s)
static_mpfr_float	1.1 (5.47s)	1.1 (9.09s)
mpf_float	**1.0** (4.82s)	**1.0**(8.07s)
cpp_dec_float	1.8 (8.54s)	2.6 (20.66s)
mpfr_class	1.3 (6.28s)	1.2(10.06s)
mpreal	2.0 (9.54s)	1.7 (14.08s)

Table 9. Non-Central T Distribution Performance

Library	50 Decimal Digits
mpfr_float	1.3 (263.27s)
static_mpfr_float	1.2 (232.88s)
mpf_float	**1.0** (195.73s)
cpp_dec_float	1.9 (366.38s)
mpfr_class	1.5 (286.94s)
mpreal	2.0 (388.70s)

Test code was compiled with Microsoft Visual Studio 2010 with all optimisations turned on (/Ox), and used MPIR-2.3.0 and MP-FR-3.0.0. The tests were run on 32-bit Windows Vista machine.

Integer Real World Tests

The first set of tests measure the times taken to execute the multiprecision part of the Voronoi-diagram builder from Boost.Polygon. The tests mainly create a large number of temporaries "just in case" multiprecision arithmetic is required, for comparison, also included in the tests is Boost.Polygon's own partial-multiprecision integer type which was custom written for this specific task:

Integer Type	Relative Performance (Actual time in parenthesis)
polygon::detail::extended_int	1(0.138831s)
int256_t	1.19247(0.165551s)
int512_t	1.23301(0.17118s)
int1024_t	1.21463(0.168628s)
checked_int256_t	1.31711(0.182855s)
checked_int512_t	1.57413(0.218538s)
checked_int1024_t	1.36992(0.190187s)
cpp_int	1.63244(0.226632s)
mpz_int	5.42511(0.753172s)
tom_int	29.0793(4.03709s)

Note how for this use case, any dynamic allocation is a performance killer.

The next tests measure the time taken to generate 1000 128-bit random numbers and test for primality using the Miller Rabin test. This is primarily a test of modular-exponentiation since that is the rate limiting step:

Integer Type	Relative Performance (Actual time in parenthesis)
cpp_int	5.25827(0.379597s)
cpp_int (no Expression templates)	5.15675(0.372268s)
cpp_int (128-bit cache)	5.10882(0.368808s)
cpp_int (256-bit cache)	5.50623(0.397497s)
cpp_int (512-bit cache)	4.82257(0.348144s)
cpp_int (1024-bit cache)	5.00053(0.360991s)
int1024_t	4.37589(0.315897s)
checked_int1024_t	4.52396(0.326587s)
mpz_int	1(0.0721905s)
mpz_int (no Expression templates)	1.0248(0.0739806s)
tom_int	2.60673(0.188181s)
tom_int (no Expression templates)	2.64997(0.191303s)

It's interesting to note that expression templates have little effect here - perhaps because the actual expressions involved are relatively trivial in this case - so the time taken for multiplication bond and division tends to dominate. Also note how increasing the internal cache size used by cpp_int is quite effective in this case in cutting out memory allocations altogether - cutting about a third off the total

runtime. Finally the much quicker times from GMP and tommath are down to their much better modular-exponentiation algorithms (GMP's is about 5x faster). That's an issue which needs to be addressed in a future release for cpp_int.

Test code was compiled with Microsoft Visual Studio 2010 with all optimisations turned on (/Ox), and used MPIR-2.3.0 and MP-FR-3.0.0. The tests were run on 32-bit Windows Vista machine.

Float Algorithm Performance

Note that these tests are carefully designed to test performance of the underlying algorithms and not memory allocation or variable copying. As usual, performance results should be taken with a healthy dose of scepticism, and real-world performance may vary widely depending upon the specifics of the program. In each table relative times are given first, with the best performer given a score of 1. Total actual times are given in brackets, measured in seconds for 500000 operations.

Table 10. Operator +

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	**1** (0.0575156s)	**1** (0.0740086s)	**1** (0.219073s)
gmp_float	2.45065 (0.14095s)	2.01398 (0.149052s)	1.09608 (0.240122s)
mpfr_float	2.6001 (0.149546s)	2.12079 (0.156957s)	1.09078 (0.23896s)

Table 11. Operator +(int)

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	1.46115 (0.0855392s)	2.60353 (0.114398s)	3.62562 (0.264905s)
gmp_float	**1** (0.0585424s)	**1** (0.0439398s)	**1** (0.0730648s)
mpfr_float	2.40441 (0.14076s)	3.2877 (0.144461s)	2.40379 (0.175632s)

Table 12. Operator +(unsigned long long)

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	**1** (0.118146s)	**1** (0.144714s)	**1** (0.315639s)
gmp_float	4.5555 (0.538213s)	3.83096 (0.554395s)	1.95079 (0.615745s)
mpfr_float	5.74477 (0.678719s)	4.85295 (0.702291s)	2.70354 (0.853342s)

Table 13. Operator +=(unsigned long long)

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	**1** (0.101188s)	**1** (0.122394s)	**1** (0.251975s)
gmp_float	5.199 (0.526079s)	4.39327 (0.537712s)	2.42151 (0.610159s)
mpfr_float	6.08318 (0.615547s)	5.18525 (0.634645s)	3.1022 (0.781677s)

Table 14. Operator -

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	**1** (0.0895163s)	**1** (0.129248s)	1.5088 (0.374512s)
gmp_float	1.72566 (0.154474s)	1.22567 (0.158415s)	**1** (0.248219s)
mpfr_float	1.83764 (0.164499s)	1.34284 (0.173559s)	1.00226 (0.248781s)

Table 15. Operator -(int)

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	**1** (0.105285s)	**1** (0.142741s)	**1** (0.278718s)
gmp_float	2.34437 (0.246828s)	1.28814 (0.183871s)	1.00731 (0.280754s)
mpfr_float	2.8032 (0.295136s)	2.09178 (0.298582s)	1.25213 (0.34899s)

Table 16. Operator -(unsigned long long)

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	**1** (0.13719s)	**1** (0.184428s)	**1** (0.344212s)
gmp_float	4.0804 (0.559791s)	3.06776 (0.565781s)	2.07736 (0.715053s)
mpfr_float	5.10114 (0.699828s)	3.88684 (0.716843s)	2.50074 (0.860784s)

Table 17. Operator -=(unsigned long long)

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	**1** (0.100984s)	**1** (0.123148s)	**1** (0.246181s)
gmp_float	5.68353 (0.573944s)	4.68636 (0.577116s)	2.6958 (0.663655s)
mpfr_float	6.19738 (0.625834s)	5.18544 (0.638577s)	3.18738 (0.784673s)

Table 18. Operator *

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	1.03667 (0.284251s)	1.30576 (0.536527s)	1.44686 (4.81057s)
gmp_float	**1** (0.274196s)	**1** (0.410891s)	**1** (3.32484s)
mpfr_float	1.24537 (0.341477s)	1.15785 (0.475749s)	1.1796 (3.92199s)

Table 19. Operator *(int)

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	3.97453 (0.240262s)	9.91222 (0.463473s)	50.7926 (4.36527s)
gmp_float	**1** (0.0604505s)	**1** (0.0467577s)	**1** (0.0859431s)
mpfr_float	2.56974 (0.155342s)	3.56312 (0.166603s)	3.22964 (0.277565s)

Table 20. Operator *(unsigned long long)

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	**1** (0.331877s)	1.01058 (0.586122s)	6.688 (4.7931s)
gmp_float	1.72433 (0.572266s)	**1** (0.579987s)	**1** (0.716672s)
mpfr_float	2.5553 (0.848047s)	1.74987 (1.0149s)	1.80403 (1.2929s)

Table 21. Operator *=(unsigned long long)

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	**1** (0.321397s)	1.00772 (0.574887s)	6.65946 (4.7468s)
gmp_float	1.77419 (0.570218s)	**1** (0.570482s)	**1** (0.712791s)
mpfr_float	2.62172 (0.842611s)	1.77691 (1.01369s)	1.77511 (1.26528s)

Table 22. Operator /

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	2.96096 (4.00777s)	4.53244 (7.86435s)	6.11936 (51.5509s)
gmp_float	**1** (1.35354s)	**1** (1.73512s)	**1** (8.42422s)
mpfr_float	1.30002 (1.75963s)	1.39045 (2.41261s)	1.66762 (14.0484s)

Table 23. Operator /(int)

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	8.60726 (1.8181s)	15.4122 (3.67479s)	34.5119 (24.729s)
gmp_float	1.24394 (0.262756s)	**1** (0.238433s)	**1** (0.716536s)
mpfr_float	**1** (0.211229s)	1.12178 (0.26747s)	1.02237 (0.732562s)

Table 24. Operator /(unsigned long long)

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	2.10976 (1.97569s)	3.73601 (3.9133s)	11.3085 (25.4533s)
gmp_float	**1** (0.936452s)	**1** (1.04746s)	**1** (2.25081s)
mpfr_float	1.3423 (1.257s)	1.51575 (1.58768s)	3.31513 (7.46175s)

Table 25. Operator /=(unsigned long long)

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	2.17401 (1.96883s)	3.79591 (3.8965s)	11.2328 (25.2606s)
gmp_float	**1** (0.905621s)	**1** (1.0265s)	**1** (2.24882s)
mpfr_float	1.37953 (1.24933s)	1.53073 (1.57129s)	3.30546 (7.43339s)

Table 26. Operator construct

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	**1** (0.00929804s)	**1** (0.0268321s)	**1** (0.0310685s)
gmp_float	30.8781 (0.287106s)	7.59969 (0.203916s)	6.51873 (0.202527s)
mpfr_float	23.5296 (0.218779s)	8.11058 (0.217624s)	7.16325 (0.222552s)

Table 27. Operator construct(unsigned)

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	**1** (0.0603971s)	**1** (0.0735485s)	**1** (0.116464s)
gmp_float	3.91573 (0.236498s)	2.88171 (0.211945s)	1.81075 (0.210887s)
mpfr_float	4.90052 (0.295977s)	4.01118 (0.295017s)	2.62005 (0.305141s)

Table 28. Operator construct(unsigned long long)

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	**1** (0.0610288s)	**1** (0.0759005s)	**1** (0.118511s)
gmp_float	8.26247 (0.504249s)	6.69042 (0.507806s)	4.32819 (0.51294s)
mpfr_float	10.1593 (0.620013s)	8.45884 (0.64203s)	5.51472 (0.653557s)

Table 29. Operator str

Backend	50 Bits	100 Bits	500 Bits
cpp_dec_float	2.95848 (0.0223061s)	3.33461 (0.033471s)	3.0159 (0.132732s)
gmp_float	**1** (0.00753971s)	**1** (0.0100374s)	**1** (0.0440106s)
mpfr_float	1.25424 (0.00945658s)	1.24943 (0.012541s)	1.09428 (0.0481601s)

Test code was compiled with Microsoft Visual Studio 2010 with all optimisations turned on (/Ox), and used MPIR-2.3.0 and MP-FR-3.0.0. The tests were run on 32-bit Windows Vista machine.

Integer Algorithm Performance

Note that these tests are carefully designed to test performance of the underlying algorithms and not memory allocation or variable copying. As usual, performance results should be taken with a healthy dose of scepticism, and real-world performance may vary widely depending upon the specifics of the program. In each table relative times are given first, with the best performer given a score of 1. Total actual times are given in brackets, measured in seconds for 500000 operations.

Table 30. Operator +

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.23704 (0.0274266s)	1.09358 (0.0383278s)	1.26645 (0.0558828s)	1.32188 (0.0916899s)
cpp_int(fixed)	1.62044 (0.0359271s)	1.5277 (0.053543s)	1.73059 (0.076363s)	1.71537 (0.118983s)
gmp_int	1.87515 (0.0415741s)	1.21699 (0.042653s)	1.15599 (0.0510088s)	**1** (0.0693631s)
tommath_int	**1** (0.0221711s)	**1** (0.035048s)	**1** (0.0441255s)	1.04441 (0.0724435s)

Table 31. Operator +(int)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	**1** (0.0155377s)	**1** (0.0209523s)	**1** (0.0306377s)	**1** (0.043125s)
cpp_int(fixed)	1.31904 (0.0204948s)	1.76211 (0.0369203s)	1.52941 (0.0468577s)	1.60412 (0.0691778s)
gmp_int	1.96204 (0.0304855s)	2.02569 (0.0424428s)	2.11505 (0.0648002s)	2.65993 (0.114709s)
tommath_int	14.0654 (0.218543s)	10.8239 (0.226786s)	7.76691 (0.23796s)	6.10039 (0.263079s)

Table 32. Operator +(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	**1** (0.026624s)	**1** (0.0291407s)	**1** (0.0373209s)	**1** (0.0464919s)
cpp_int(fixed)	1.31378 (0.034978s)	1.54897 (0.045138s)	1.53649 (0.0573431s)	1.27833 (0.0594319s)
gmp_int	25.5775 (0.680974s)	24.0117 (0.699717s)	19.5633 (0.730121s)	16.8939 (0.785432s)
tommath_int	19.4694 (0.518354s)	18.4246 (0.536907s)	14.7715 (0.551288s)	12.3637 (0.574812s)

Table 33. Operator +=(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.18405 (0.0196905s)	1.22304 (0.0206476s)	1.25861 (0.0217397s)	1.29525 (0.0220829s)
cpp_int(fixed)	**1** (0.0166298s)	**1** (0.0168822s)	**1** (0.0172728s)	**1** (0.0170492s)
gmp_int	39.9082 (0.663668s)	39.4584 (0.666147s)	38.5504 (0.665873s)	39.2231 (0.668722s)
tommath_int	30.6219 (0.509238s)	30.4135 (0.513447s)	30.9077 (0.533863s)	32.3086 (0.550835s)

Table 34. Operator -

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.06986 (0.0296064s)	**1** (0.0381508s)	1.05932 (0.053186s)	1.1766 (0.0844721s)
cpp_int(fixed)	1.3304 (0.0368163s)	1.44506 (0.0551303s)	1.4431 (0.0724545s)	1.57255 (0.112898s)
gmp_int	1.48072 (0.0409761s)	1.19003 (0.0454007s)	1.0794 (0.0541942s)	**1** (0.0717934s)
tommath_int	**1** (0.0276731s)	1.10891 (0.0423057s)	**1** (0.0502076s)	1.08479 (0.0778811s)

Table 35. Operator -(int)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	**1** (0.0147372s)	**1** (0.0170001s)	**1** (0.0232882s)	**1** (0.0310734s)
cpp_int(fixed)	1.4267 (0.0210256s)	1.98887 (0.0338109s)	1.83788 (0.0428009s)	1.81269 (0.0563264s)
gmp_int	2.07504 (0.0305803s)	2.40928 (0.0409579s)	2.58711 (0.0602493s)	3.26438 (0.101435s)
tommath_int	13.5424 (0.199577s)	12.1793 (0.207048s)	9.28855 (0.216314s)	7.49327 (0.232842s)

Table 36. Operator -(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	**1** (0.0277377s)	**1** (0.0296807s)	**1** (0.0372392s)	**1** (0.0455855s)
cpp_int(fixed)	1.19867 (0.0332484s)	1.48639 (0.0441169s)	1.43253 (0.0533464s)	1.27697 (0.0582111s)
gmp_int	24.1794 (0.670683s)	22.9073 (0.679904s)	18.8758 (0.702922s)	16.5837 (0.755975s)
tommath_int	18.149 (0.503413s)	17.4116 (0.516787s)	14.0411 (0.52288s)	11.8237 (0.538987s)

Table 37. Operator -=(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.26896 (0.0203467s)	1.25722 (0.0206147s)	1.36108 (0.0225485s)	1.18351 (0.0226161s)
cpp_int(fixed)	**1** (0.0160342s)	**1** (0.0163971s)	**1** (0.0165667s)	**1** (0.0191094s)
gmp_int	41.1339 (0.659547s)	40.3982 (0.662411s)	39.925 (0.661425s)	34.636 (0.661874s)
tommath_int	31.1543 (0.499533s)	31.0303 (0.508806s)	30.7699 (0.509756s)	27.7054 (0.529434s)

Table 38. Operator *

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.11839 (0.0757577s)	1.61061 (0.207951s)	1.4501 (0.696912s)	1.72796 (2.64108s)
cpp_int(fixed)	1.01115 (0.0684934s)	1.28687 (0.166152s)	**1** (0.480595s)	**1** (1.52844s)
gmp_int	**1** (0.0677384s)	**1** (0.129113s)	1.09011 (0.523902s)	1.03374 (1.58s)
tommath_int	1.6322 (0.110562s)	2.71751 (0.350866s)	2.05222 (0.986288s)	2.0644 (3.15531s)

Table 39. Operator *(int)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.01611 (0.0229536s)	1.12175 (0.0298152s)	1.16413 (0.0416439s)	1.31747 (0.0666043s)
cpp_int(fixed)	1.30215 (0.0294152s)	1.669 (0.0443606s)	1.72395 (0.0616701s)	1.88315 (0.095202s)
gmp_int	**1** (0.0225897s)	**1** (0.0265791s)	**1** (0.0357725s)	**1** (0.0505547s)
tommath_int	10.8281 (0.244603s)	10.1516 (0.26982s)	8.76424 (0.313519s)	8.04364 (0.406644s)

Table 40. Operator *(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	**1** (0.0570721s)	**1** (0.0856141s)	**1** (0.143279s)	**1** (0.252785s)
cpp_int(fixed)	1.10857 (0.0632686s)	1.2951 (0.110878s)	1.20827 (0.173121s)	1.18463 (0.299456s)
gmp_int	12.0605 (0.68832s)	8.13434 (0.696415s)	5.21762 (0.747577s)	3.11601 (0.787681s)
tommath_int	10.0524 (0.57371s)	7.33116 (0.627651s)	4.85202 (0.695193s)	3.35808 (0.848871s)

Table 41. Operator *=(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	111.27 (7.43118s)	67.7078 (7.34138s)	43.3851 (7.4075s)	25.3089 (7.55455s)
cpp_int(fixed)	**1** (0.0667848s)	**1** (0.108427s)	**1** (0.170738s)	**1** (0.298493s)
gmp_int	46.3718 (3.09693s)	28.4639 (3.08626s)	18.1719 (3.10264s)	10.5223 (3.14083s)
tommath_int	276.674 (18.4776s)	169.146 (18.34s)	108.491 (18.5236s)	63.3261 (18.9024s)

Table 42. Operator /

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	2.68035 (0.595251s)	2.04702 (0.707471s)	1.62314 (0.921536s)	1.43112 (1.38811s)
cpp_int(fixed)	**1** (0.222079s)	**1** (0.34561s)	**1** (0.567748s)	**1** (0.969945s)
gmp_int	3.79283 (0.842308s)	2.73668 (0.945824s)	1.86649 (1.05969s)	1.32141 (1.2817s)
tommath_int	13.2531 (2.94324s)	11.2054 (3.87271s)	9.83293 (5.58262s)	13.0164 (12.6252s)

Table 43. Operator /(int)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	4.06026 (0.225473s)	3.45732 (0.340049s)	3.00195 (0.547957s)	2.80587 (0.978029s)
cpp_int(fixed)	2.43766 (0.135367s)	2.56264 (0.252052s)	2.44011 (0.445402s)	2.38009 (0.829617s)
gmp_int	**1** (0.0555316s)	**1** (0.0983563s)	**1** (0.182534s)	**1** (0.348566s)
tommath_int	35.9988 (1.99907s)	27.1024 (2.66569s)	21.8333 (3.98531s)	25.8066 (8.99528s)

Table 44. Operator /(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.50505 (0.705756s)	1.39347 (1.58556s)	2.63348 (3.57438s)	4.75451 (8.52733s)
cpp_int(fixed)	**1** (0.468925s)	1.12378 (1.27869s)	2.29966 (3.12128s)	4.4844 (8.04288s)
gmp_int	2.17234 (1.01866s)	**1** (1.13785s)	**1** (1.35728s)	**1** (1.79352s)
tommath_int	4.74612 (2.22557s)	2.70088 (3.07319s)	3.65634 (4.96268s)	6.79408 (12.1853s)

Table 45. Operator /=(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.76281 (0.0574966s)	1.76471 (0.0604224s)	1.56085 (0.0716403s)	1.31422 (0.124043s)
cpp_int(fixed)	**1** (0.0326164s)	**1** (0.0342393s)	**1** (0.0458981s)	**1** (0.0943852s)
gmp_int	20.2862 (0.661664s)	19.4043 (0.664389s)	14.4881 (0.664976s)	7.14238 (0.674135s)
tommath_int	32.9555 (1.07489s)	30.1525 (1.0324s)	22.8324 (1.04796s)	11.7456 (1.10861s)

Table 46. Operator %

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.8501 (0.364131s)	1.46527 (0.476653s)	1.27509 (0.689738s)	1.20064 (1.11769s)
cpp_int(fixed)	**1** (0.196817s)	**1** (0.325301s)	**1** (0.540932s)	**1** (0.930916s)
gmp_int	3.2533 (0.640305s)	2.15441 (0.700832s)	1.47898 (0.800029s)	1.07439 (1.00016s)
tommath_int	15.3501 (3.02116s)	12.1106 (3.9396s)	11.0689 (5.98752s)	13.5535 (12.6172s)

Table 47. Operator %(int)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.82761 (0.104331s)	2.01496 (0.202512s)	2.10004 (0.389523s)	2.17252 (0.768097s)
cpp_int(fixed)	1.78851 (0.102099s)	1.96844 (0.197838s)	2.02956 (0.376451s)	2.07257 (0.73276s)
gmp_int	**1** (0.057086s)	**1** (0.100505s)	**1** (0.185483s)	**1** (0.353552s)
tommath_int	36.3018 (2.07233s)	26.3075 (2.64402s)	21.9525 (4.07183s)	25.6759 (9.07775s)

Table 48. Operator construct

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.40211 (0.0026854s)	**1** (0.00278639s)	**1** (0.00322813s)	**1** (0.0027185s)
cpp_int(fixed)	**1** (0.00191526s)	1.40721 (0.00392103s)	1.90346 (0.00614463s)	2.14621 (0.00583447s)
gmp_int	98.705 (0.189046s)	68.9726 (0.192184s)	58.8994 (0.190135s)	70.0525 (0.190438s)
tommath_int	105.602 (0.202255s)	74.1994 (0.206748s)	63.6455 (0.205456s)	76.8935 (0.209035s)

Table 49. Operator construct(unsigned)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.73436 (0.00348927s)	**1** (0.00263476s)	**1** (0.0027009s)	**1** (0.00318651s)
cpp_int(fixed)	**1** (0.00201185s)	1.36851 (0.0036057s)	2.07362 (0.00560064s)	1.66856 (0.00531688s)
gmp_int	97.2414 (0.195635s)	76.3759 (0.201232s)	72.7396 (0.196462s)	63.8129 (0.20334s)
tommath_int	210.112 (0.422713s)	162.652 (0.42855s)	158.33 (0.427634s)	134.626 (0.428987s)

Table 50. Operator construct(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	2.34403 (0.00739542s)	1.66376 (0.00713834s)	1.22989 (0.0074969s)	1.23708 (0.00711417s)
cpp_int(fixed)	**1** (0.00315501s)	**1** (0.00429049s)	**1** (0.00609561s)	**1** (0.0057508s)
gmp_int	222.866 (0.703144s)	164.331 (0.705059s)	115.363 (0.70321s)	122.347 (0.703596s)
tommath_int	218.681 (0.689941s)	163.796 (0.702765s)	114.57 (0.698376s)	122.422 (0.704027s)

Table 51. Operator gcd

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.16358 (2.74442s)	1.39847 (8.11559s)	1.64677 (22.2518s)	1.95096 (64.4961s)
cpp_int(fixed)	**1** (2.35859s)	1.30986 (7.60133s)	1.67681 (22.6577s)	2.0895 (69.0758s)
gmp_int	1.03392 (2.4386s)	**1** (5.80319s)	**1** (13.5124s)	**1** (33.0586s)
tommath_int	5.25978 (12.4057s)	4.4619 (25.8932s)	4.15577 (56.1542s)	3.91192 (129.323s)

Table 52. Operator powm

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	2.50722 (2.91621s)	3.5561 (13.406s)	4.37066 (73.483s)	4.88831 (473.91s)
cpp_int(fixed)	1.93385 (2.24931s)	3.18107 (11.9922s)	4.20753 (70.7403s)	4.8158 (466.88s)
gmp_int	**1** (1.16313s)	**1** (3.76986s)	**1** (16.8128s)	**1** (96.9476s)
tommath_int	1.44081 (1.67584s)	1.8794 (7.08507s)	2.19115 (36.8394s)	2.17186 (210.557s)

Table 53. Operator str

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.17175 (0.00160006s)	1.41999 (0.00329476s)	1.40856 (0.00813784s)	1.52964 (0.0229767s)
cpp_int(fixed)	**1** (0.00136554s)	**1** (0.00232027s)	**1** (0.00577741s)	1.14754 (0.0172372s)
gmp_int	1.50501 (0.00205515s)	1.52968 (0.00354926s)	1.01989 (0.0058923s)	**1** (0.015021s)
tommath_int	12.2161 (0.0166816s)	16.9577 (0.0393463s)	18.7474 (0.108311s)	22.7368 (0.341528s)

Table 54. Operator |

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	**1** (0.0301617s)	**1** (0.0423404s)	**1** (0.0522358s)	**1** (0.0813156s)
cpp_int(fixed)	1.0638 (0.0320861s)	1.22566 (0.0518951s)	1.28515 (0.0671305s)	1.16118 (0.094422s)
gmp_int	1.76553 (0.0532514s)	1.51489 (0.0641408s)	1.70708 (0.0891706s)	1.77346 (0.14421s)
tommath_int	4.37637 (0.131999s)	3.46212 (0.146587s)	2.91875 (0.152463s)	4.19621 (0.341217s)

Table 55. Operator |(int)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	**1** (0.0289129s)	**1** (0.0351119s)	**1** (0.0406779s)	**1** (0.0525891s)
cpp_int(fixed)	1.06091 (0.030674s)	1.25979 (0.0442336s)	1.36194 (0.0554009s)	1.37438 (0.0722772s)
gmp_int	4.92854 (0.142498s)	4.34687 (0.152627s)	3.71442 (0.151095s)	2.981 (0.156768s)
tommath_int	10.9847 (0.317598s)	9.37065 (0.329021s)	8.53651 (0.347248s)	11.2155 (0.589813s)

Table 56. Operator ^

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	**1** (0.0305149s)	**1** (0.04217s)	**1** (0.0525977s)	**1** (0.0816632s)
cpp_int(fixed)	1.01544 (0.0309861s)	1.24872 (0.0526585s)	1.26661 (0.066621s)	1.15965 (0.0947007s)
gmp_int	1.64675 (0.0502505s)	1.47181 (0.0620663s)	1.66038 (0.0873322s)	1.67895 (0.137108s)
tommath_int	4.30668 (0.131418s)	3.45859 (0.145849s)	2.91462 (0.153303s)	4.15538 (0.339342s)

Table 57. Operator ^(int)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.01566 (0.0296088s)	**1** (0.0356634s)	**1** (0.0401898s)	**1** (0.0514097s)
cpp_int(fixed)	**1** (0.0291524s)	1.2393 (0.0441976s)	1.38556 (0.0556856s)	1.38899 (0.0714075s)
gmp_int	4.68027 (0.136441s)	4.15243 (0.14809s)	3.74237 (0.150405s)	3.0483 (0.156712s)
tommath_int	10.919 (0.318314s)	9.16311 (0.326788s)	8.62554 (0.346659s)	11.6212 (0.597442s)

Table 58. Operator &

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.0346 (0.0303431s)	**1** (0.0427309s)	**1** (0.0535587s)	1.06945 (0.0828084s)
cpp_int(fixed)	**1** (0.0293284s)	1.10435 (0.04719s)	1.05262 (0.0563769s)	**1** (0.0774309s)
gmp_int	1.86057 (0.0545675s)	1.58432 (0.0676995s)	1.69164 (0.0906018s)	1.86625 (0.144505s)
tommath_int	4.4157 (0.129506s)	3.60396 (0.154s)	2.95985 (0.158525s)	4.4032 (0.340944s)

Table 59. Operator &(int)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	1.05874 (0.038946s)	**1** (0.0483903s)	**1** (0.063842s)	**1** (0.100361s)
cpp_int(fixed)	**1** (0.0367853s)	1.05827 (0.0512099s)	1.09114 (0.0696605s)	1.09432 (0.109826s)
gmp_int	3.92298 (0.144308s)	2.99447 (0.144903s)	2.228 (0.14224s)	1.42296 (0.142809s)
tommath_int	8.79208 (0.323419s)	7.02288 (0.339839s)	5.65271 (0.36088s)	6.27104 (0.629365s)

Table 60. Operator <<

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	**1** (0.0248801s)	1.23196 (0.04s)	**1** (0.0424149s)	**1** (0.060157s)
cpp_int(fixed)	1.08931 (0.027102s)	1.40572 (0.0456418s)	1.3475 (0.0571542s)	1.24573 (0.0749397s)
gmp_int	1.05561 (0.0262636s)	**1** (0.0324686s)	1.09914 (0.0466199s)	1.16315 (0.0699719s)
tommath_int	1.60497 (0.0399319s)	2.13048 (0.0691737s)	2.31219 (0.0980712s)	2.74695 (0.165248s)

Table 61. Operator >>

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_int	**1** (0.0213349s)	1.02127 (0.0295019s)	**1** (0.0327116s)	1.13168 (0.0433804s)
cpp_int(fixed)	1.13514 (0.0242181s)	1.16938 (0.0337803s)	1.46999 (0.0480859s)	1.60077 (0.061362s)
gmp_int	1.26614 (0.0270129s)	**1** (0.0288873s)	1.42219 (0.0465221s)	**1** (0.0383329s)
tommath_int	12.0066 (0.25616s)	10.2837 (0.297067s)	9.99696 (0.327017s)	16.0943 (0.616942s)

Test code was compiled with Microsoft Visual Studio 2010 with all optimisations turned on (/Ox), and used MPIR-2.3.0 and MP-FR-3.0.0. The tests were run on 32-bit Windows Vista machine.

Linux x86_64 results are broadly similar, except that libtommath performs much better there.

Rational Type Performance

Note that these tests are carefully designed to test performance of the underlying algorithms and not memory allocation or variable copying. As usual, performance results should be taken with a healthy dose of scepticism, and real-world performance may vary widely depending upon the specifics of the program. In each table relative times are given first, with the best performer given a score of 1. Total actual times are given in brackets, measured in seconds for 500000 operations.

Table 62. Operator +

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	5.89417 (18.4116s)	6.87256 (47.4698s)	6.65008 (107.715s)	6.53801 (256.244s)
mpq_rational	**1** (3.1237s)	**1** (6.90715s)	**1** (16.1975s)	**1** (39.1929s)

Table 63. Operator +(int)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	3.62367 (2.46488s)	4.18291 (2.94603s)	4.726 (3.74866s)	6.1388 (5.56817s)
mpq_rational	**1** (0.680215s)	**1** (0.704303s)	**1** (0.7932s)	**1** (0.907046s)

Table 64. Operator +(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	1.1527 (2.6378s)	1.31751 (3.09863s)	1.58996 (4.00714s)	2.15642 (5.75702s)
mpq_rational	**1** (2.28837s)	**1** (2.35189s)	**1** (2.52028s)	**1** (2.66971s)

Table 65. Operator +=(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	1.18436 (2.7059s)	1.32279 (3.11099s)	1.61398 (4.05389s)	2.20048 (5.84623s)
mpq_rational	**1** (2.2847s)	**1** (2.35183s)	**1** (2.51174s)	**1** (2.6568s)

Table 66. Operator -

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	5.81893 (18.3457s)	6.82209 (47.1928s)	6.64143 (107.498s)	6.51362 (255.137s)
mpq_rational	**1** (3.15277s)	**1** (6.91765s)	**1** (16.1859s)	**1** (39.1698s)

Table 67. Operator -(int)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	3.72441 (2.48756s)	4.27663 (2.98713s)	4.62109 (3.72114s)	6.17605 (5.56503s)
mpq_rational	**1** (0.667908s)	**1** (0.698479s)	**1** (0.805252s)	**1** (0.901066s)

Table 68. Operator -(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	1.15627 (2.63239s)	1.32096 (3.12092s)	1.61044 (4.00106s)	2.19378 (5.7644s)
mpq_rational	**1** (2.27663s)	**1** (2.36262s)	**1** (2.48445s)	**1** (2.62761s)

Table 69. Operator -=(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	1.1984 (2.73444s)	1.34141 (3.15698s)	1.64159 (4.06997s)	2.23017 (5.88108s)
mpq_rational	**1** (2.28174s)	**1** (2.35348s)	**1** (2.47929s)	**1** (2.63706s)

Table 70. Operator *

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	5.4306 (32.5882s)	6.91805 (89.9436s)	6.94556 (207.307s)	6.88704 (492.151s)
mpq_rational	**1** (6.00084s)	**1** (13.0013s)	**1** (29.8475s)	**1** (71.4604s)

Table 71. Operator *(int)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	2.12892 (2.51376s)	2.47245 (3.07841s)	2.86832 (3.93619s)	3.94086 (6.02565s)
mpq_rational	**1** (1.18077s)	**1** (1.24508s)	**1** (1.3723s)	**1** (1.52902s)

Table 72. Operator *(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	1.32254 (5.43565s)	1.56078 (6.73163s)	1.97701 (9.32522s)	2.85404 (15.1573s)
mpq_rational	**1** (4.11002s)	**1** (4.313s)	**1** (4.71682s)	**1** (5.31082s)

Table 73. Operator *=(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	6.29806 (58.1188s)	6.30556 (59.5076s)	6.3385 (62.1007s)	6.55345 (67.6905s)
mpq_rational	**1** (9.22804s)	**1** (9.43733s)	**1** (9.79739s)	**1** (10.329s)

Table 74. Operator /

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	4.4269 (66.8031s)	6.40103 (173.527s)	6.32347 (348.193s)	6.61148 (824.063s)
mpq_rational	**1** (15.0903s)	**1** (27.1093s)	**1** (55.0637s)	**1** (124.641s)

Table 75. Operator /(int)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	1.78772 (2.50984s)	2.10623 (3.10606s)	2.46986 (3.99358s)	3.37428 (5.96678s)
mpq_rational	**1** (1.40393s)	**1** (1.4747s)	**1** (1.61693s)	**1** (1.76831s)

Table 76. Operator /(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	1.29695 (5.45454s)	1.55248 (6.85353s)	1.93237 (9.28765s)	2.75211 (14.8541s)
mpq_rational	**1** (4.20568s)	**1** (4.41458s)	**1** (4.80635s)	**1** (5.39734s)

Table 77. Operator /=(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	6.19401 (58.4278s)	6.20135 (59.643s)	6.21327 (62.0338s)	6.40576 (67.6778s)
mpq_rational	**1** (9.43295s)	**1** (9.61774s)	**1** (9.98407s)	**1** (10.5652s)

Table 78. Operator construct

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	**1** (0.00978288s)	**1** (0.0100574s)	**1** (0.0101393s)	**1** (0.0101847s)
mpq_rational	39.1516 (0.383015s)	38.3523 (0.385725s)	37.5812 (0.381048s)	37.6007 (0.382953s)

Table 79. Operator construct(unsigned)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	**1** (0.0548151s)	**1** (0.0557542s)	**1** (0.055825s)	**1** (0.0552808s)
mpq_rational	7.21073 (0.395257s)	7.1016 (0.395944s)	7.02046 (0.391917s)	7.16881 (0.396297s)

Table 80. Operator construct(unsigned long long)

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	**1** (0.0605156s)	**1** (0.0616657s)	**1** (0.0592056s)	**1** (0.0603081s)
mpq_rational	35.1604 (2.12775s)	34.7575 (2.14335s)	35.7232 (2.11502s)	35.0437 (2.11342s)

Table 81. Operator str

Backend	128 Bits	256 Bits	512 Bits	1024 Bits
cpp_rational	5.48898 (0.0208949s)	8.49668 (0.0546688s)	10.107 (0.121897s)	10.5339 (0.310584s)
mpq_rational	**1** (0.0038067s)	**1** (0.00643413s)	**1** (0.0120606s)	**1** (0.0294843s)

Test code was compiled with Microsoft Visual Studio 2010 with all optimisations turned on (/Ox), and used MPIR-2.3.0 and MPFR-3.0.0. The tests were run on 32-bit Windows Vista machine.

Roadmap

History

1.54

- **Breaking change** renamed `rational_adapter` to `rational_adaptor`.

- Add support for MPFI.

- Add logged_adaptor.

- Add support for 128-bit floats via GCC's `__float128` or Intel's `_Quad` data types.

- Add support for user-defined literals in cpp_int, improve `constexpr` support.

- Fixed bug in integer division of `cpp_int` that results in incorrect sign of `cpp_int` when both arguments are small enough to fit in a `double_limb_type`. See 8126.

- Fixed bug in subtraction of a single limb in `cpp_int` that results in incorrect value when the result should have a 0 in the last limb: 8133.

- Fixed bug in `cpp_int` where division of 0 by something doesn't get zero in the result: 8160.

- Fixed bug in some transcendental functions that caused incorrect return values when variables are reused, for example with `a = pow(a, b)`. See 8326.

- Fixed some assignment operations in the mpfr and gmp backends to be safe if the target has been moved from: 8667.

- Fixed bug in `cpp_int` that gives incorrect answer for 0%N for large N: 8670.

- Fixed set_precision in mpfr backend so it doesn't trample over an existing value: 8692.

1.53

- First Release.

- Fix bug in cpp_int division.

- Fix issue #7806.

Post review changes

- Non-expression template operators further optimised with rvalue reference support.

- Many functions made `constexp`.

- Differentiate between explicit and implicit conversions in the number constructor.

- Removed "mp_" prefix from types.

- Allowed mixed precision arithmetic.

- Changed ExpressionTemplates parameter to class `number` to use enumerated values rather than true/false.

- Changed ExpressionTemplate parameter default value to use a traits class so that the default value depends on the backend used.

- Added support for fused-multiply-add/subtract with GMP support.

- Tweaked expression template unpacking to use fewer temporaries when the LHS also appears in the RHS.

- Refactored `cpp_int_backend` based on review comments with new template parameter structure.

- Added additional template parameter to `mpfr_float_backend` to allow stack-based allocation.

- Added section on mixed precision arithmetic, and added support for operations yielding a higher precision result than either of the arguments.

- Added overloads of integer-specific functions for built in integer types.

Pre-review history

- 2011-2012, John Maddock adds an expression template enabled front end to Christopher's code, and adds support for other backends.

- 2011, Christopher Kormanyos publishes the decimal floating point code under the Boost Software Licence. The code is published as: "Algorithm 910: A Portable C++ Multiple-Precision System for Special-Function Calculations", in ACM TOMS, {VOL 37, ISSUE 4, (February 2011)} (C) ACM, 2011.

- 2002-2011, Christopher Kormanyos develops the all C++ decimal arithmetic floating point code.

TODO

More a list of what *could* be done, rather than what *should* be done (which may be a much smaller list!).

- Add back-end support for libdecNumber.

- Add an adaptor back-end for complex number types.

- Add a back-end for MPFR interval arithmetic.

- Add better multiplication routines (Karatsuba, FFT etc) to cpp_int_backend.

- Add assembly level routines to cpp_int_backend.

- Add an all C++ Boost licensed binary floating point type.

- Can ring types (exact floating point types) be supported? The answer should be yes, but someone needs to write it, the hard part is IO and binary-decimal conversion.

- Should there be a choice of rounding mode (probably MPFR specific)?

- We can reuse temporaries in multiple subtrees (temporary caching).

- cpp_dec_float should round to nearest.

- A 2's complement fixed precision int that uses exactly N bits and no more.

Things requested in review:

- The performances of mp_number<a_trivial_adaptor<float>, false>respect to float and mp_number<a_trivial_adaptor<int>, false> and int should be given to show the cost of using the generic interface (Mostly done, just need to update docs to the latest results).

- Should we provide min/max overloads for expression templates? (Not done - we can't overload functions declared in the std namespace :-( ).

- The rounding applied when converting must be documented (Done).

- Document why we don't abstract out addition/multiplication algorithms etc. (done - FAQ)

- Document why we don't use proto (compile times) (Done).

- We can reuse temporaries in multiple subtrees (temporary caching) Moved to TODO list.

- Emphasise in the docs that ET's may reorder operations (done 2012/10/31).

- Document what happens to small fixed precision cpp_int's (done 2012/10/31).

- The use of bool in template parameters could be improved by the use of an enum class which will be more explicit. E.g `enum class expression_template {disabled, enabled}; enum class sign {unsigned, signed};` (Partly done 2012/09/15, done 2012/10/31).

- Each back-end should document the requirements it satisfies (not currently scheduled for inclusion: it's deliberately an implementation detail, and "optional" requirements are optimisations which can't be detected by the user). Not done: this is an implementation detail, the exact list of requirements satisfied is purely an optimization, not something the user can detect.

- A backend for an overflow aware integers (done 2012/10/31).

- IIUC convert_to is used to emulate in c++98 compilers C++11 explicit conversions. Could the explicit conversion operator be added on compilers supporting it? (Done 2012/09/15).

- The front-end should make the differences between implicit and explicit construction (Done 2012/09/15).

- The tutorial should add more examples concerning implicit or explicit conversions. (Done 2012/09/15).

- The documentation must explain how move semantics helps in this domain and what the backend needs to do to profit from this optimization. (Done 2012/09/15).

- The documentation should contain Throws specification on the mp_number and backend requirements operations. (Done 2012/09/15).

- The library interface should use the noexcept (BOOST_NOEXCEPT, ...) facilities (Done 2012/09/15).

- It is unfortunate that the generic mp_number front end can not make use constexpr as not all the backends can ensure this (done - we can go quite a way).

- literals: The library doesn't provide some kind of literals. I think that the mp_number class should provide a way to create literals if the backend is able to. (Done 2012/09/15).

- The ExpresionTemplate parameter could be defaulted to a traits class for more sensible defaults (done 2012/09/20).

- In a = exp1 op exp2 where a occurs inside one of exp1 or exp2 then we can optimise and eliminate one more temporary (done 2012/09/20).

Pre-Review Comments

- Make fixed precision orthogonal to Allocator type in cpp_int. Possible solution - add an additional MaxBits template argument that defaults to 0 (meaning keep going till no more space/memory). Done.

- Can ring types (exact floating point types) be supported? The answer should be yes, but someone needs to write it (Moved to TODO list).

- Should there be a choice of rounding mode (probably MPFR specific)? Moved to TODO list.

- Make the exponent type for cpp_dec_float a template parameter, maybe include support for big-integer exponents. Open question - what should be the default - int32_t or int64_t? (done 2012/09/06)

- Document the size requirements of fixed precision ints (done 2012/09/15).

- Document std lib function accuracy (done 2012/09/15).

- Be a bit clearer on the effects of sign-magnitude representation of cpp_int - min == -max etc - done.

- Document cpp_dec_float precision, rounding, and exponent size (done 2012/09/06).

- Can we be clearer in the docs that mixed arithmetic doesn't work (no longer applicable as of 2012/09/06)?

- Document round functions behaviour better (they behave as in C++11) (added note 2012/09/06).

- Document limits on size of cpp_dec_float (done 2012/09/06).

- Add support for fused multiply add (and subtract). GMP mpz_t could use this (done 2012/09/20).

FAQ

Why do I get compiler errors when passing a `number` to a template function?	Most likely you are actually passing an expression template type to the function and template-argument-deduction deduces the "wrong" type. Try casting the arguments involving expressions to the actual number type, or as a last resort turning off expression template support in the number type you are using.
When is expression template support a performance gain?	As a general rule, expression template support adds a small runtime overhead creating and unpacking the expression templates, but greatly reduces the number of temporaries created. So it's most effective in improving performance when the cost of creating a temporary is high: for example when creating a temporary involves a memory allocation. It is least effective (and may even be a dis-optimisation) when temporaries are cheap: for example if the number type is basically a thin wrapper around a native arithmetic type. In addition, since the library makes extensive use of thin inline wrapper functions, turning on compiler optimization is essential to achieving high performance.
Do expression templates reorder operations?	Yes they do, sometimes quite radically so, if this is a concern then they should be turned off for the number type you are using.
I can't construct my number type from *some other type*, but the docs indicate that the conversion should be allowed, what's up?	Some conversions are *explicit*, that includes construction from a string, or constructing from any type that may result in loss of precision (for example constructing an integer type from a float).
Why do I get an exception thrown (or the program crash due to an uncaught exception) when using the bitwise operators on a checked `cpp_int`?	Bitwise operations on negative values (or indeed any signed integer type) are unspecified by the standard. As a result any attempt to carry out a bitwise operation on a negative checked-integer will result in a `std::range_error` being thrown.
Why do I get compiler errors when trying to use the complement operator?	Use of the complement operator on signed types is problematic as the result is unspecified by the standard, and is further complicated by the fact that most extended precision integer types use a sign-magnitude representation rather than the 2's complement one favored by most native integer types. As a result the complement operator is deliberately disabled for checked `cpp_int`'s. Unchecked `cpp_int`'s give the same valued result as a 2's complement type would, but not the same bit-pattern.
Why can't I negate an unsigned type?	The unary negation operator is deliberately disabled for unsigned integer types as its use would almost always be a programming error.
Why doesn't the library use proto?	A very early version of the library did use proto, but compile times became too slow for the library to be usable. Since the library only required a tiny fraction of what proto has to offer anyway, a lightweight expression template mechanism was used instead. Compile times are still too slow...
Why not abstract out addition/multiplication algorithms?	This was deemed not to be practical: these algorithms are intimately tied to the actual data representation used.

Acknowledgements

This library would not have happened without:

- Christopher Kormanyos' C++ decimal number code.

- Paul Bristow for patiently testing, and commenting on the library.

- All the folks at GMP, MPFR and libtommath, for providing the "guts" that makes this library work.

- "The Art Of Computer Programming", Donald E. Knuth, Volume 2: Seminumerical Algorithms, Third Edition (Reading, Massachusetts: Addison-Wesley, 1997), xiv+762pp. ISBN 0-201-89684-2

Indexes

Function Index

A

abs
   number, 60
add
   Mixed Precision Arithmetic, 57
   number, 60
   TODO, 124
assign
   number, 60
assign_components
   Optional Requirements on the Backend Type, 76

B

bit_flip
   number, 60
bit_set
   number, 60
bit_test
   number, 60
bit_unset
   number, 60

C

compare
   number, 60

D

data
   float128, 23
   gmp_float, 19
   gmp_int, 12
   gmp_rational, 39
   mpfi_float, 34
   mpfr_float, 21
default_precision
   number, 60
divide_qr
   number, 60

E

empty
   mpfi_float, 34
eval_acos
   Optional Requirements on the Backend Type, 76
eval_add
   Compulsory Requirements on the Backend type., 76
   Optional Requirements on the Backend Type, 76
eval_asin
   Optional Requirements on the Backend Type, 76
eval_atan
   Optional Requirements on the Backend Type, 76

Class Index

G

gmp_float
   gmp_float, 19, 74
gmp_int
   gmp_int, 12, 73
gmp_rational
   gmp_rational, 39

I

is_explicitly_convertible
   Internal Support Code, 76
is_lossy_conversion
   Internal Support Code, 76
is_number
   number, 60
is_number_expression
   number, 60
is_restricted_conversion
   Internal Support Code, 76
is_signed_number
   Internal Support Code, 76
is_unsigned_number
   Internal Support Code, 76

L

logged_adaptor
   logged_adaptor, 42

M

mpfi_float_backend
   mpfi_float, 34
mpfr_float_backend
   mpfr_float, 21, 75
   mpfr_float_backend, 75

N

number
   number, 60
number_category
   number, 60

T

tommath_int
   tom_int, 13, 74

Typedef Index

C

checked_cpp_int
   cpp_int, 9, 72
checked_cpp_rational
   cpp_int, 9, 72
checked_cpp_rational_backend
   cpp_int, 9, 72
checked_int1024_t

F

I

L

M

Index

A

abs
    number, 60
add
    Mixed Precision Arithmetic, 57
    number, 60
    TODO, 124
assign
    number, 60
assign_components
    Optional Requirements on the Backend Type, 76

B

Bit Operations
    r, 16
bit_flip
    number, 60
bit_set
    number, 60
bit_test
    number, 60
bit_unset
    number, 60
BOOST_MP_DEFINE_SIZED_CPP_INT_LITERAL
    Literal Types and constexpr Support, 54

C

Calculating an Integral
    mp_type, 30
checked_cpp_int
    cpp_int, 9, 72
checked_cpp_rational
    cpp_int, 9, 72
checked_cpp_rational_backend
    cpp_int, 9, 72
checked_int1024_t
    cpp_int, 9, 72
checked_int128_t
    cpp_int, 9, 72
checked_int256_t
    cpp_int, 9, 72
checked_int512_t
    cpp_int, 9, 72
checked_uint1024_t
    cpp_int, 9, 72
checked_uint128_t
    cpp_int, 9, 72
checked_uint256_t
    cpp_int, 9, 72
checked_uint512_t
    cpp_int, 9, 72
compare
    number, 60
component_type

L

M

N

O

T

U

V

Z