multiprecision Users Manual.

Christopher Kormanyos

Table of Contents

Warning

This documentation is very much under construction!

Important

This is NOT yet an official Boost library.

Note

Comments and suggestions (even bugs!) to Christopher Kormanyos (at) e_float (dot) yahoo (dot) com

A PDF version of this manual that is printer-friendly is also available.

Introduction

About multiprecision

There are many multiple precision (MP) packages available to the scientific and engineering community. Each package has individual strengths within its range of application. However, most MP packages lack a uniform interface for high precision algorithm design. They also offer little or no special function support. Also, many MP packages have limited portability. There is no portable standalone C++ system which offers a wide variety of high precision special functions and handles large function parameters.

The multiprecision system (previously called extended float) not only addresses these weaknesses but also significantly advances MP technology. It uses several MP packages and provides a uniform C++ interface for high precision algorithm design, independent of the underlying MP implementation. In addition, multiprecision supports a large collection of high performance MP functions which are entirely portable, solidly designed and can be used with any suitably prepared MP type. Furthermore, the multiprecision system provides software interfaces for seamless interoperability with other high level languages. No MP system other than multiprecision offers such a high degree of portability, such a wide range of functions, and such a rich selection of powerful interoperabilities. The multiprecision library can be used for a variety of purposes such as high precision function evaluation, numerical verification of compilers, function libraries or other computational systems as well as investigations of compiler quality, optimization methods and computer hardware.

Multiprecision is unique because it is designed from the ground up utilizing generic and object oriented design methods to create an efficient and flexible numerical software architecture [1], [2], [36].

The standard containers and algorithms of the C++ STL and TR1 are consistently used to reduce programmatic complexity [3], [14], [15], [20]. Static and dynamic polymorphism are used to implement a standardized interface to MP types allowing for the interchangeable use of different MP implementations.

Multiprecision is written in the C++ language making broad use of the C++ core language, much of the STL and some of TR1, as specified in ISO/IEC 14882:2003 and ISO/IEC 19768:2007 [14, super 15]. It is emphasized that the C++ compiler must closely adhere to these standards in order to successfully compile and link multiprecision. The source codes have been implemented according to safe programming and reliability standards originating from the automotive industry [26, 27]. A great effort has been invested in advanced C++ optimization techniques in order to improve system performance. Generic and object oriented programming methods have been used to create an efficient and flexible numerical software architecture. In addition, consistent use of standard containers and algorithms of the STL and TR1 [20, 3] has significantly reduced and evenly distributed the computational complexities of the entire program.

...

Configurations

The multiprecision system supports a variety of configurations using several multiprecision classes as well as other libraries and systems. These are shown in Table 1. Details about the capabilities and dependencies of the configurations are also included in the table below.

The Multiprecision System Architecture

The multiprecision system architecture is robust and flexible. With this architecture, both the integration of other MP types as well as the addition of more functions and interoperabilities can be done with ease. The system architecture is shown in Figure 1. It has four layers and two additional blocks, the test block and the tools block. Layers 1-4 have successively increasing levels of abstraction. They build up very high level functionalities in a stable, stepwise fashion.

Note

that *multiprecision* is not only the name of the system but also the name of several multiprecision classes.

[1] James O. Coplien. Advanced C++ Programming Styles and Idioms. Addison Wesley, Reading Massachusetts, 1992.
[2] MISRA. MISRA-C++ 2008: Guidelines for the Use of the C++ Language in Critical Systems. MISRA Consortium, MISRA-C++, 2008.

- Layer 1, the low level MP layer, ensures that each implementation-dependent, possibly nonportable MP implementation is suitably prepared to conform with the C++ class requirements of Layer 2. Each individual MP type is encapsulated within a specific multiprecision C++ class, each one of which defined within its own unique namespace. For example, classes such as efx::multiprecision, gmp::multiprecision and others are implemented. Each of these classes is derived from a common abstract base class called ::multiprecision base. The abstract base class defines public and pure virtual functions which implement arithmetic primitives such as self-multiplication, self-compare, special numbers like NaN, and string operations. These arithmetic primitives fulfill the requirements necessary for elementary mathematics. They simultaneously conform with Layer 2. Thus, via inheritance and implementation of virtual functions, each individual multiprecision class supports elementary mathematics and also complies with Layer 2.

Some MP types include their own versions of various functions. Layer 1 accommodates this with the "has-its-own"-mechanism. This mechanism allows the C++ interface of a given MP type to use the MP s own algorithm for a particular function. It uses virtual Boolean functions prefixed with "has its own". For example, has its own sin returns true in order to use the MPs own implementation of sin(x), x2R. The performance of a specific MP class can be optimized by selectively activating these functions. MPFR [11] has its own implementations of most elementary functions and several higher transcendental functions. The elementary functions are quite efficient and these are, in fact, used to optimize the performance of mpfr::multiprecision.

- Layer 2 implements the uniform C++ interface. The multiprecision type from layer 1, which will be used in the project configuration, is selected with a compile-time option. The functions of this multiprecision class are used to implement all arithmetic operations, numeric limits and basic I/O mechanisms. Thus, layer 2 provides a uniform C++ interface which is generic and fully equipped with all the basic functions necessary for high level MP algorithm design in a C++ environment.

- Layer 3 is the C++ mathematical layer. It adds the class ef complex, i.e. the complex data type. This layer uses both the selected multiprecision type as well as ef complex to implement multiprecision's rich collection of elementary functions and higher transcendental functions.

- Layer 4, the interoperability user layer, exposes all of the functions and capabilities of layers 2 and 3 to other high level languages. Marshaling techniques [34] are used to create managed C++ classes and wrapper functions which embody the full functionality of layers 2 and 3. These are compiled to a single CLR assembly which can be used with all Microsoft® CLR languages including C#, managed C++/CLI, IronPython, etc. Compatibility with the Microsoft®.NET Framework 3.5 has been tested. Another interoperability employs the boost.python library [1] to expose the functionality of layers 2 and 3 to Python. Compatibilities with Python 2.6.4 and boost 1.39 have been tested.

Another layer 4 interoperability targets Mathematica ®. A sparse architecture has been developed to create a generic interface for interacting with computer algebra systems. The compatibility of this interface with Mathematica® 7.1 has been tested. The interoperabilities of layer 4 are very powerful mechanisms based on highly advanced programming techniques. They can be used for very high level designs such as scripting, rapid algorithm prototyping and result visualization.

diagram

The test block contains several hundred automatically generated test files which have been specifically designed to test all algorithms and convergence regions of the entire multiprecision system. The test block includes an automatic test execution system and an automatic test case generator. This block allows for fully reproducible automated testing of the system.

The tool block contains a variety of utilities and examples. The utilities predominantly consist of generic templates for standard mathematical operations such as numerical differentiation, root finding, recursive quadrature, etc. The examples show practical, non-trivial uses of the multiprecision system involving both high level algorithm design as well as interoperability.

The multiprecision architecture exemplifies how layered design can be leveraged to find the right granularity to distribute a very large computational complexity among smaller constituents which have manageable software scope. For example, there are vast software distances between the handoptimized assembler routines of GNU MP [15] and, for example, the Hurwitz zeta function, or a high level GUI in C#. The multiprecision architecture elegantly navigates these distances to build up high level functionalities in a controlled, stepwise fashion.

The multiprecision system architecture is a significant technological milestone in MP programming technology. While other MP packages do sometimes provide a specialized C++ interface for their own specific implementations, they are mostly incompatible with each other. However, multiprecision's uniform C++ layer creates a generic interface which can be used with any underlying MP type. Assuming that a given MP type can be brought into conformance with layer 2, it can be used in a portable fashion with all

of multiprecision's capabilities including arithmetic, elementary functions, special functions, interoperabilities, automatic tests, utilities, and additional user-created extensions.

Examples

Using Multiprecision with a pre-built library

Building Multiprecision Library

To build a multiprecision library, see folder

../../build/build_libraries/

and use the jamfile.v2 therein, by changing directory to your version of *multiprecision/libs/multiprecision/build/build_libraries*

and run the jamfile using

```
b2 > multiprecision_lib_build.log
```

This will create a static library in that directory (you can copy or install it elsewhere if you prefer, for example boost-1.nnn/stage/lib).

Whereever you place it, the library must be visible to the example programs. (For Microsoft Visual, either add the folder to to the VC++ Directories, Library Directories and add `libboost_multiprecision` to the an Linker, Input, Additional Dependencies).

Using a Library

This section shows a few simple aspects of using multiprecision with a pre-built library.

Multiprecision example using a pre-built library.

Select the mp_float back-end big-number type by defining BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_TYPE_xxx, for example, as a compiler defined preprocessor macro.

For example `#define BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_TYPE_EFX;` in MS Project Properties, Preprocessor, Preprocessor definitions or select backend type EFX (which has the Boost licence) below with the #define below.

```
#define BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_TYPE_EFX
```

Note

If you have other files that use mp_float, they will all need this #define, so a single project-wide definition may be more convenient.

Warning

Ensure that MS language extensions are NOT disabled.

Includes are needed to use multiprecision, mp_float.hpp for arithmetic and trig functions, and mp_float_functions.hpp to be able to use various constants.

```
#include <boost/multiprecision/mp_float.hpp>
#include <boost/multiprecision/mp_float_functions.hpp> // for constants.
```

Note

It may seem (and be) simplest to declare global use of the namespace `boost::multiprecision` but this risks conflicts with names that appear in other namespaces, like std. So it is safer to limit the scope to within user classes and functions, or to specify explicitly the functions, either with a using statement, or, more verbosely, at the point of use, as various shown below.

To start, we display the precision currently available (a compile-time constant), and the number of possibly significant and guaranteed digits from `std::numeric_limits` (which is fully specified for `mp_float`).

```
int main()
{
  using boost::multiprecision::mp_float;

  cout << "BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_DIGITS10 = " << BOOST_MULTIPRECI↵
SION_BACKEND_MP_FLOAT_DIGITS10 << endl;

  cout.precision(std::numeric_limits<mp_float>::max_digits10);
  cout << "std::numeric_limits<multiprecision>::max_digits10  = " << std::numeric_lim↵
its<mp_float>::max_digits10 << endl;
  cout << "std::numeric_limits<multiprecision>::digits10  = " << std::numeric_limits<mp_float>::di↵
gits10 << endl;
```

We then construct an mp_float from an integer (10), and also another mp_float using a constant function `boost::multiprecision::hundred();` and use it to calculate a googol. We then try (but fail) to calculate a googleplex, but the result is correctly reported as infinity because is exceeds even the monster range of multiprecision (at default precision), shown by using numeric_limits

```
std::numeric_limits<mp_float>::max()
= 1.000000000000000000000000000000000000000000000000000000000000000000000000000000e+003063937869882635617
```

```
mp_float float_ten(10);

mp_float float_hundred = boost::multiprecision::hundred();

mp_float googol;
googol = pow(float_ten, float_hundred);
cout << "googol = " << showpoint << googol << endl; //  = ↵
1.000000000000000000000000000000000000000000000000000000000000000000000000000000e+100

mp_float googolplex;
googolplex = pow(float_ten, googol);
cout << "googolplex = " << showpoint << googolplex << endl; // inf!

mp_float mp_float_max = boost::multiprecision::value_max();
cout << "std::numeric_limits<mp_float>::max() = " << std::numeric_limits<mp_float>::max() << endl;


//cout << "boost::multiprecision::value_max() = " << showpoint << mp_float_max << endl;
```

Many other constants are available as functions. It would be verbose and inconvenient to have to fully specify each use fully, so `using` statements are sensible.

```
using boost::multiprecision::pi; // Convenient to permit just pi().

mp_float my_pi;
my_pi = pi();
cout << "boost::multiprecision::pi(); = " << pi() << endl;
cout << " 4/3 pi(); = " << 4 * pi()/3 << endl;
```

Other constants can be derived with no loss of precision, but as the next examples show, there are dragons awaiting the unwary!

The first is a C++ danger, familiar using integers and built-in floating-point types.

```
double r = 41/47; // zero!
```

We **must** use a floating-point type constant (41.0) for the numerator

```
double r = 41.0/47;
```

but there are further dangers using multiprecision and builtin floating-point types double... from the risk of often silent but catastrophic loss of precision converting from the much lower precision built-in type. So if we innocently write

```
mp_float r = 41./47;
```

Warning

It **appears** to get the right result, but in fact loses precision from converting from double to mp_float, typically near the the 17th decimal digit.

So one needs to be very careful to always work with `mp_float`, for example by static_casting.

```
mp_float r = static_cast<mp_float>(41)/47;
```

```
using boost::multiprecision::pi; // Convenient to permit just pi().

mp_float my_pi;
my_pi = pi();
cout << "boost::multiprecision::pi(); = " << pi() << endl;
cout << " 4/3 pi(); = " << 4 * pi()/3 << endl;
```

The full output from this example is

```
using_library_example.cpp
 Generating code
 Finished generating code
 using_library_example.vcxproj -> I:\boost-sandbox\multiprecision\libs\multiprecision\build\Re↵
lease\using_library_example.exe
 BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_DIGITS10 = 50
 std::numeric_limits<multiprecision>::max_digits10  = 58
 std::numeric_limits<multiprecision>::digits10  = 50
 googol = 1.0000000000000000000000000000000000000000000000000000000000000000000000000000e+100
 googolplex = inf
 std::numeric_lim↵
its<mp_float>::max() = 1.0000000000000000000000000000000000000000000000000000000000000000000000000000e+003063937869882635617
 boost::multiprecision::pi(); = 3.1415926535897932384626433832795028841971693993751058209 75
  4/3 pi(); = 4.1887902047863909846168578443726705122628925325001410946 33
 tan (41/47) = 1.1909692408064234338549953586439165516621022462967354785 73
```

The full source code of this examples is at using_library_example.cpp.

Using multiprecision *headeronly*

Some users like to avoid pre-built libraries (static .LIB or synamic shared .DLL.

Since Boost.Multiprecision has separate declarations in header files and definitions in sources files, at Header files, it is necessary include many `.cpp` source files as well `.hpp` files.

The header files needed in the user program are

```
#include <boost/multiprecision/mp_float.hpp>
#include <boost/multiprecision/mp_float_functions.hpp> // for constants.
```

The full set of source files is

```
backends/float/mp_float.cpp
backends/float/mp_float_base.cpp
backends/float/efx/mp_float_efx.cpp

functions/constants/constants.cpp
functions/elementary/elementary_complex.cpp
functions/elementary/elementary_hyper_g.cpp
functions/elementary/elementary_math.cpp
functions/elementary/elementary_trans.cpp
functions/elementary/elementary_trig.cpp
functions/gamma/factorial.cpp
functions/gamma/factorial2.cpp
functions/gamma/gamma.cpp
functions/gamma/pochhammer.cpp
functions/integer/bernoulli_b.cpp
functions/integer/prime.cpp
functions/integer/prime_factor.cpp
functions/zeta/zeta.cpp
utility/util_digit_scale.cpp
utility/util_power_j_pow_x.cpp
utility/util_timer.cpp
```

These need to be added to the MS Project Files.

Here Be Dragons

Pitfalls in Constructing `mp_float` variables

Construction of `mp_float` from all `int` types is always exact.

```
mp_float i(12345);
cout << "mp_float i(12345); = " << i << endl; // mp_float i(12345); = 12345
```

Construction from `integer` literal string is always exact.

```
mp_float is("12345"); // Construction from integer string is always exact.
cout << "mp_float i(\"12345\"); = " << is << endl; // mp_float i("12345"); = 12345
```

Construction from `double` with integral value is always exact.

```
mp_float id(12345.);
cout << "mp_float i(12345.); = " << id << endl; //   mp_float i(12345.); = 12345
```

Construction from `float` with integral value is always exact.

```
mp_float ifl(12345.F); // Construction from float with integral value is always exact.
cout << "mp_float i(12345.F); = " << ifl << endl; //   mp_float i(12345.F); = 12345
```

The examples above work because the value is integral, with zero fractional part, but if the fraction part is **not zero**, we are about to make a mistake.

```
mp_float ↵
v1234567890 = 1.23456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890;
//        v1234567890 = 1.23456789012345669043213547411141917109489440917968 75
//                                            ^
```

See the catastrophic loss of accuracy starting at the 17th decimal digit! (`std::numeric_limits<double>::max_digits10` = 17 for the common 64-bit representation).

When the uncertainty of value is known to be more than the precision of the type (typical when it is a measurement - few are known to 17 decimal digit accuracy) there may be no loss of information : for a mathematical constant, every digit is meaningful.

Pitfalls in Constructing `mp_float` fractions

. Creating fractions from integer numerator in C++ is always dangerous because the result is probably NOT integral! This is a mistake that everyone writing C++ makes at least once. The numerator should always be a floating-point type, built-in like `double`, or `mp_float`, whatever the desired result type, also always a floating-point type.

```
const double seventh = 1/7; // seventh == Zero!
```

Using `mp_float` is no different, and the dangers are the same.

```
const mp_float z = 1/7; // zero!
cout << "const mp_float r = 1/7; = " << z << endl;
```

The simplest way is to ensure that the numerator is of type `mp_float`.

```
const mp_float r = mp_float(1)/7; //   const mp_float r = 1/7; = 0
cout << "const mp_float r = mp_float(1)/7 = " << r << endl;
// const mp_float r = mp_float(1)/7 = 0.142857142857142857
```

The full output from this example is

```
dragons_example.vcxproj -> I:\boost-sandbox\multiprecision\libs\multiprecision\build\Re↵
lease\dragons_example.exe
mp_float i(12345); = 12345
mp_float i("12345"); = 12345
mp_float i(12345.); = 12345
mp_float i(12345.F); = 12345
const mp_float r = 1/7; = 0
const mp_float r = mp_float(1)/7 = 0.142857142857142857
```

Document Conventions

- **Tutorials** are listed in the *Table of Contents* and include many examples that should help you get started quickly.
- **Source code** of the many *Examples* will often be your quickest starting point.
- **Reference section** prepared using Doxygen will provide the function and class signatures, but there is also an *index* of these.
- The main *index* will also help, especially if you know a word describing what it does, without needing to know the exact name chosen for the function.

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

- C++ Code is in `fixed width font` and is syntax-highlighted in color.
- Other code is in `teletype fixed-width font`.
- Replaceable text that you will need to supply is in *`italics`*.
- If a name refers to a free function, it is specified like this: `free_function()`; that is, it is in *`code font`* and its name is followed by `()` to indicate that it is a free function.
- If a name refers to a class template, it is specified like this: `class_template<>`; that is, it is in code font and its name is followed by <> to indicate that it is a class template.
- If a name refers to a function-like macro, it is specified like this: `MACRO()`; that is, it is uppercase in code font and its name is followed by `()` to indicate that it is a function-like macro. Object-like macros appear without the trailing `()`.
- Names that refer to *concepts* in the generic programming sense are specified in CamelCase.
- Many code snippets assume an implicit namespace, for example, `std::` or `boost::checks`.
- If you have a feature request, or if it appears that the implementation is in error, please check the TODO section first, as well as the rationale section.

If you do not find your idea/complaint, please reach the author either through the Boost development list, or email the author(s) direct .

Admonishments

Note

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

Tip

These blocks contain information that you may find helpful while coding.

Important

These contain information that is imperative to understanding a concept. Failure to follow suggestions in these blocks will probably result in undesired behavior. Read all of these you find.

Warning

Failure to heed this will lead to incorrect, and very likely undesired, results.

Acknowledgements

- Paul A. Bristow produced a prototype of this Quickbook documentation of Multiprecision.
- Docbook authors and maintainers.
- BoostBook (an extension of DocBook) was developed by Douglas Gregor.
- Quickbook was developed by Joel de Guzman and Eric Niebler, and now maintained by Daniel James.
- The link to Doxygen was pioneered by Doug Gregor, and developed by Steven Watanabe.
- Rendering to PDF was made to work by John Maddock.
- Improvements to the tool chain, especially XSLT, by Daniel James and a welcome speedup and other enhancements by Steven Watanabe.
- RenderX kindly provide free use of XEP to render the PDF files from XML.
- Automatic Indexing was developed by John Maddock.
- Doxygen is maintained by Dimitri van Heesch. It was developed by many people who are acknowledged at the Doxygen site.

References

- James O. Coplien. Advanced C++ Programming Styles and Idioms. Addison Wesley, Reading Massachusetts, 1992.
- MISRA. MISRA-C++ 2008: Guidelines for the Use of the C++ Language in Critical Systems. MISRA Consortium, MISRA-C++, 2008.

Design Rationale

1. **Precision - Compile-time Versus versus Run-time.**
   After consultation on the Boost list, it was decided that the best compromise was to fix precision compile-time, rather than allow the choice to be made at run-time.
   A major factor was the sheer difficulty of achieving a run-time solution, but also the risk of performance penalty, longer compile times, risk of errors from complexity, greater difficulty of testing.

2. **Choice of backend** It was considered important to permit a choice of backend. Although GMP/MPFR is the *Gold Standard*, the restricted licence terms make it useless for any commerical applications, so it was deemed essential to provide a Boost license backend, even if its performance was not quite as good.

3. **Mechanisn of choosing backend** It was decided that using a macro BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_TYPE_xxx was the best way to achieve this.

4. **Default precision of 50** was chosen to be greater than any floating-point hardware, but not so big that printing a single value would overflow typical line length. (

5. **Upper limit of 300 decimal digits** was chosen TODO.

6. **Base-10 radix** was chosen over base-2 radix because TODO.

7. **Automatic Conversions and Construction**.
   It was initially planned to enforce explicit conversions, for example, conversion from integer and floating-point to mp_float. However, integer conversions are exact unless they cause overflow, a separate issue. Floating-point conversion carries a risk of loss of precision. However, in practice, it was found that the burden on the user in providing explicit conversion, for example by `static_casting` was too great for the benefit in safety. A specific example was that, for conformance with Boost.Math, boost::multiprecision::mp_float required automatic conversion to/from any and all built-in types, including in association with add, sub, mul, div. So it was decided to rely on documentation to warning (loudly and repeatedly) and the danger of construction and conversion from built-in `float`, `double` and `long double`.

8. **Separation of definitions from declarations** Definitions are all placed in source files of type `.cpp` in a subdirectory `/src`. Header files containing declarations are placed as usual in `boost/multiprecision/*.hpp`. This makes it possible to use header-only (if slightly less convenient, because the .cpp files have to included as well) or with building a multiprecision library. Separating source may reduce compile time.

Appendix: Tickets

Report and view bugs and features by adding a ticket at Boost.Trac.

Existing open tickets for this library alone can be viewed here. Existing tickets for this library - including closed ones - can be viewed here.

Version Info

Last edit to Quickbook file multiprecision.qbk was at 04:53:19 PM on 2011-Oct-31.

Tip

This version information should appear on the pdf version (but is redundant on html where the last revised date is on the bottom line of the home page).

Warning

Home page "Last revised" is GMT, not local time. Last edit date is local time.

Caution

It does not give the last edit date of other included .qbk files, so may mislead!

multiprecision C++ Reference

Header <boost/multiprecision/mp_complex.hpp>

```
namespace boost {
  namespace multiprecision {
    class mp_complex;
    bool operator!=(const mp_complex & u, const mp_complex & v);
    mp_complex operator*(const mp_complex & u, const mp_complex & v);
    mp_complex operator*(const mp_complex & u, const mp_float & v);
    mp_complex operator*(const mp_float & u, const mp_complex & v);
    mp_complex operator*(const mp_complex & z, const char n);
    mp_complex operator*(const mp_complex & z, const signed char n);
    mp_complex operator*(const mp_complex & z, const signed short n);
    mp_complex operator*(const mp_complex & z, const int n);
    mp_complex operator*(const mp_complex & z, const signed long n);
    mp_complex operator*(const mp_complex & z, const signed long long n);
    mp_complex operator*(const mp_complex & z, const unsigned char n);
    mp_complex operator*(const mp_complex & z, const wchar_t n);
    mp_complex operator*(const mp_complex & z, const unsigned short n);
    mp_complex operator*(const mp_complex & z, const unsigned int n);
    mp_complex operator*(const mp_complex & z, const unsigned long n);
    mp_complex operator*(const mp_complex & z, const unsigned long long n);
    mp_complex operator*(const mp_complex & z, const float f);
    mp_complex operator*(const mp_complex & z, const double d);
    mp_complex operator*(const mp_complex & z, const long double ld);
    mp_complex operator*(const char n, const mp_complex & v);
    mp_complex operator*(const signed char n, const mp_complex & v);
    mp_complex operator*(const signed short n, const mp_complex & v);
    mp_complex operator*(const int n, const mp_complex & v);
    mp_complex operator*(const signed long n, const mp_complex & v);
    mp_complex operator*(const signed long long n, const mp_complex & v);
    mp_complex operator*(const unsigned char n, const mp_complex & v);
    mp_complex operator*(const wchar_t n, const mp_complex & v);
    mp_complex operator*(const unsigned short n, const mp_complex & v);
    mp_complex operator*(const unsigned int n, const mp_complex & v);
    mp_complex operator*(const unsigned long n, const mp_complex & v);
    mp_complex operator*(const unsigned long long n, const mp_complex & v);
    mp_complex operator*(const float f, const mp_complex & v);
    mp_complex operator*(const double d, const mp_complex & v);
    mp_complex operator*(const long double ld, const mp_complex & v);
    mp_complex & operator*=(mp_complex &, const char);
    mp_complex & operator*=(mp_complex &, const signed char);
    mp_complex & operator*=(mp_complex &, const int);
    mp_complex & operator*=(mp_complex &, const signed long long);
    mp_complex & operator*=(mp_complex &, const unsigned char);
    mp_complex & operator*=(mp_complex &, const wchar_t);
    mp_complex & operator*=(mp_complex &, const unsigned long);
    mp_complex & operator*=(mp_complex &, const unsigned long long);
    mp_complex & operator*=(mp_complex &, const float);
    mp_complex & operator*=(mp_complex &, const double);
    mp_complex & operator*=(mp_complex &, const long double);
    mp_complex & operator*=(mp_complex &, const mp_float &);
    mp_complex & operator*=(mp_complex & z, const signed short n);
    mp_complex & operator*=(mp_complex & z, const signed long n);
    mp_complex & operator*=(mp_complex & z, const unsigned short n);
    mp_complex & operator*=(mp_complex & z, const unsigned int n);
    mp_complex & operator+(mp_complex & u);
    const mp_complex & operator+(const mp_complex & u);
    mp_complex operator+(const mp_complex & u, const mp_complex & v);
    mp_complex operator+(const mp_complex & u, const mp_float & v);
```

```
mp_complex operator+(const mp_float & u, const mp_complex & v);
mp_complex operator+(const mp_complex & z, const char n);
mp_complex operator+(const mp_complex & z, const signed char n);
mp_complex operator+(const mp_complex & z, const signed short n);
mp_complex operator+(const mp_complex & z, const int n);
mp_complex operator+(const mp_complex & z, const signed long n);
mp_complex operator+(const mp_complex & z, const signed long long n);
mp_complex operator+(const mp_complex & z, const unsigned char n);
mp_complex operator+(const mp_complex & z, const wchar_t n);
mp_complex operator+(const mp_complex & z, const unsigned short n);
mp_complex operator+(const mp_complex & z, const unsigned int n);
mp_complex operator+(const mp_complex & z, const unsigned long n);
mp_complex operator+(const mp_complex & z, const unsigned long long n);
mp_complex operator+(const mp_complex & z, const float f);
mp_complex operator+(const mp_complex & z, const double d);
mp_complex operator+(const mp_complex & z, const long double ld);
mp_complex operator+(const char n, const mp_complex & v);
mp_complex operator+(const signed char n, const mp_complex & v);
mp_complex operator+(const signed short n, const mp_complex & v);
mp_complex operator+(const int n, const mp_complex & v);
mp_complex operator+(const signed long n, const mp_complex & v);
mp_complex operator+(const signed long long n, const mp_complex & v);
mp_complex operator+(const unsigned char n, const mp_complex & v);
mp_complex operator+(const wchar_t n, const mp_complex & v);
mp_complex operator+(const unsigned short n, const mp_complex & v);
mp_complex operator+(const unsigned int n, const mp_complex & v);
mp_complex operator+(const unsigned long n, const mp_complex & v);
mp_complex operator+(const unsigned long long n, const mp_complex & v);
mp_complex operator+(const float f, const mp_complex & v);
mp_complex operator+(const double d, const mp_complex & v);
mp_complex operator+(const long double ld, const mp_complex & v);
mp_complex operator++(mp_complex & u, int);
mp_complex & operator+=(mp_complex &, const char);
mp_complex & operator+=(mp_complex &, const signed char);
mp_complex & operator+=(mp_complex &, const int);
mp_complex & operator+=(mp_complex &, const signed long long);
mp_complex & operator+=(mp_complex &, const unsigned char);
mp_complex & operator+=(mp_complex &, const wchar_t);
mp_complex & operator+=(mp_complex &, const unsigned long);
mp_complex & operator+=(mp_complex &, const unsigned long long);
mp_complex & operator+=(mp_complex &, const float);
mp_complex & operator+=(mp_complex &, const double);
mp_complex & operator+=(mp_complex &, const long double);
mp_complex & operator+=(mp_complex &, const mp_float &);
mp_complex & operator+=(mp_complex & z, const signed short n);
mp_complex & operator+=(mp_complex & z, const signed long n);
mp_complex & operator+=(mp_complex & z, const unsigned short n);
mp_complex & operator+=(mp_complex & z, const unsigned int n);
mp_complex operator-(const mp_complex & u);
mp_complex operator-(const mp_complex & u, const mp_complex & v);
mp_complex operator-(const mp_complex & u, const mp_float & v);
mp_complex operator-(const mp_float & u, const mp_complex & v);
mp_complex operator-(const mp_complex & z, const char n);
mp_complex operator-(const mp_complex & z, const signed char n);
mp_complex operator-(const mp_complex & z, const signed short n);
mp_complex operator-(const mp_complex & z, const int n);
mp_complex operator-(const mp_complex & z, const signed long n);
mp_complex operator-(const mp_complex & z, const signed long long n);
mp_complex operator-(const mp_complex & z, const unsigned char n);
mp_complex operator-(const mp_complex & z, const wchar_t n);
mp_complex operator-(const mp_complex & z, const unsigned short n);
mp_complex operator-(const mp_complex & z, const unsigned int n);
mp_complex operator-(const mp_complex & z, const unsigned long n);
```

```
mp_complex operator-(const mp_complex & z, const unsigned long long n);
mp_complex operator-(const mp_complex & z, const float f);
mp_complex operator-(const mp_complex & z, const double d);
mp_complex operator-(const mp_complex & z, const long double ld);
mp_complex operator-(const char n, const mp_complex & v);
mp_complex operator-(const signed char n, const mp_complex & v);
mp_complex operator-(const signed short n, const mp_complex & v);
mp_complex operator-(const int n, const mp_complex & v);
mp_complex operator-(const signed long n, const mp_complex & v);
mp_complex operator-(const signed long long n, const mp_complex & v);
mp_complex operator-(const unsigned char n, const mp_complex & v);
mp_complex operator-(const wchar_t n, const mp_complex & v);
mp_complex operator-(const unsigned short n, const mp_complex & v);
mp_complex operator-(const unsigned int n, const mp_complex & v);
mp_complex operator-(const unsigned long n, const mp_complex & v);
mp_complex operator-(const unsigned long long n, const mp_complex & v);
mp_complex operator-(const float f, const mp_complex & v);
mp_complex operator-(const double d, const mp_complex & v);
mp_complex operator-(const long double ld, const mp_complex & v);
mp_complex operator--(mp_complex & u, int);
mp_complex & operator-=(mp_complex &, const char);
mp_complex & operator-=(mp_complex &, const signed char);
mp_complex & operator-=(mp_complex &, const int);
mp_complex & operator-=(mp_complex &, const signed long long);
mp_complex & operator-=(mp_complex &, const unsigned char);
mp_complex & operator-=(mp_complex &, const wchar_t);
mp_complex & operator-=(mp_complex &, const unsigned long);
mp_complex & operator-=(mp_complex &, const unsigned long long);
mp_complex & operator-=(mp_complex &, const float);
mp_complex & operator-=(mp_complex &, const double);
mp_complex & operator-=(mp_complex &, const long double);
mp_complex & operator-=(mp_complex &, const mp_float &);
mp_complex & operator-=(mp_complex & z, const signed short n);
mp_complex & operator-=(mp_complex & z, const signed long n);
mp_complex & operator-=(mp_complex & z, const unsigned short n);
mp_complex & operator-=(mp_complex & z, const unsigned int n);
mp_complex operator/(const mp_complex & u, const mp_complex & v);
mp_complex operator/(const mp_complex & u, const mp_float & v);
mp_complex operator/(const mp_float & u, const mp_complex & v);
mp_complex operator/(const mp_complex & z, const char n);
mp_complex operator/(const mp_complex & z, const signed char n);
mp_complex operator/(const mp_complex & z, const signed short n);
mp_complex operator/(const mp_complex & z, const int n);
mp_complex operator/(const mp_complex & z, const signed long n);
mp_complex operator/(const mp_complex & z, const signed long long n);
mp_complex operator/(const mp_complex & z, const unsigned char n);
mp_complex operator/(const mp_complex & z, const wchar_t n);
mp_complex operator/(const mp_complex & z, const unsigned short n);
mp_complex operator/(const mp_complex & z, const unsigned int n);
mp_complex operator/(const mp_complex & z, const unsigned long n);
mp_complex operator/(const mp_complex & z, const unsigned long long n);
mp_complex operator/(const mp_complex & z, const float f);
mp_complex operator/(const mp_complex & z, const double d);
mp_complex operator/(const mp_complex & z, const long double ld);
mp_complex operator/(const char n, const mp_complex & v);
mp_complex operator/(const signed char n, const mp_complex & v);
mp_complex operator/(const signed short n, const mp_complex & v);
mp_complex operator/(const int n, const mp_complex & v);
mp_complex operator/(const signed long n, const mp_complex & v);
mp_complex operator/(const signed long long n, const mp_complex & v);
mp_complex operator/(const unsigned char n, const mp_complex & v);
mp_complex operator/(const wchar_t n, const mp_complex & v);
mp_complex operator/(const unsigned short n, const mp_complex & v);
```

```
    mp_complex operator/(const unsigned int n, const mp_complex & v);
    mp_complex operator/(const unsigned long n, const mp_complex & v);
    mp_complex operator/(const unsigned long long n, const mp_complex & v);
    mp_complex operator/(const float f, const mp_complex & v);
    mp_complex operator/(const double d, const mp_complex & v);
    mp_complex operator/(const long double ld, const mp_complex & v);
    mp_complex & operator/=(mp_complex &, const char);
    mp_complex & operator/=(mp_complex &, const signed char);
    mp_complex & operator/=(mp_complex &, const int);
    mp_complex & operator/=(mp_complex &, const signed long long);
    mp_complex & operator/=(mp_complex &, const unsigned char);
    mp_complex & operator/=(mp_complex &, const wchar_t);
    mp_complex & operator/=(mp_complex &, const unsigned long);
    mp_complex & operator/=(mp_complex &, const unsigned long long);
    mp_complex & operator/=(mp_complex &, const float);
    mp_complex & operator/=(mp_complex &, const double);
    mp_complex & operator/=(mp_complex &, const long double);
    mp_complex & operator/=(mp_complex &, const mp_float &);
    mp_complex & operator/=(mp_complex & z, const signed short n);
    mp_complex & operator/=(mp_complex & z, const signed long n);
    mp_complex & operator/=(mp_complex & z, const unsigned short n);
    mp_complex & operator/=(mp_complex & z, const unsigned int n);
    std::basic_ostream< char, std::char_traits< char > > &
    operator<<(std::basic_ostream< char, std::char_traits< char > > &,
               const mp_complex &);
    bool operator==(const mp_complex & u, const mp_complex & v);
    std::basic_istream< char, std::char_traits< char > > &
    operator>>(std::basic_istream< char, std::char_traits< char > > &,
               mp_complex &);
  }
}
```

Class mp_complex

boost::multiprecision::mp_complex

Synopsis

```
// In header: <boost/multiprecision/mp_complex.hpp>


class mp_complex {
public:
  // construct/copy/destruct
  mp_complex(const char);
  mp_complex(const signed char);
  mp_complex(const unsigned char);
  mp_complex(const wchar_t);
  mp_complex(const signed short);
  mp_complex(const unsigned short);
  mp_complex(const int);
  mp_complex(const unsigned int);
  mp_complex(const signed long);
  mp_complex(const unsigned long);
  mp_complex(const signed long long);
  mp_complex(const unsigned long long);
  mp_complex(const float);
  mp_complex(const double);
  mp_complex(const long double);
  explicit mp_complex(const char *const);
  explicit mp_complex(const std::string &);
  mp_complex();
  mp_complex(const mp_float &);
  mp_complex(const mp_float &, const mp_float &);
  mp_complex(const mp_complex &);
  mp_complex& operator=(const mp_complex &);
  mp_complex& operator=(const mp_float &);

  // public member functions
  mp_float imag(void) const;
  bool isfinite(void) const;
  bool isinf(void) const;
  bool isint(void) const;
  bool isnan(void) const;
  bool isneg(void) const;
  bool isone(void) const;
  bool ispos(void) const;
  bool iszero(void) const;
  mp_float norm(void) const;
  mp_complex & operator*=(const mp_complex &);
  const mp_complex & operator++(void);
  mp_complex & operator+=(const mp_complex &);
  const mp_complex & operator--(void);
  mp_complex & operator-=(const mp_complex &);
  mp_complex & operator/=(const mp_complex &);
  mp_float real(void) const;

  // public static functions
  static mp_float imag(const mp_complex &);
  static mp_float real(const mp_complex &);
};
```

Description

`mp_complex` public construct/copy/destruct

1. ```
   mp_complex(const char n);
   ```
   ```

2. ```
   mp_complex(const signed char n);
   ```

3. ```
   mp_complex(const unsigned char n);
   ```

4. ```
   mp_complex(const wchar_t n);
   ```

5. ```
   mp_complex(const signed short n);
   ```

6. ```
   mp_complex(const unsigned short n);
   ```

7. ```
   mp_complex(const int n);
   ```

8. ```
   mp_complex(const unsigned int n);
   ```

9. ```
   mp_complex(const signed long n);
   ```

10. ```
    mp_complex(const unsigned long n);
    ```

11. ```
    mp_complex(const signed long long n);
    ```

12. ```
    mp_complex(const unsigned long long n);
    ```

13. ```
    mp_complex(const float f);
    ```

14. ```
    mp_complex(const double d);
    ```

15. ```
    mp_complex(const long double ld);
    ```

16. ```
    explicit mp_complex(const char *const s);
    ```

17. ```
    explicit mp_complex(const std::string & str);
    ```

18. `mp_complex();`

19. `mp_complex(const mp_float & re);`

20. `mp_complex(const mp_float & re, const mp_float & im);`

21. `mp_complex(const mp_complex & z);`

22. `mp_complex& operator=(const mp_complex & v);`

23. `mp_complex& operator=(const mp_float & v);`

`mp_complex` public member functions

1. `mp_float imag(void) const;`

2. `bool isfinite(void) const;`

3. `bool isinf(void) const;`

4. `bool isint(void) const;`

5. `bool isnan(void) const;`

6. `bool isneg(void) const;`

7. `bool isone(void) const;`

8. `bool ispos(void) const;`

9. `bool iszero(void) const;`

10. `mp_float norm(void) const;`

11. ```
mp_complex & operator*=(const mp_complex & v);
```

12. ```
const mp_complex & operator++(void);
```

13. ```
mp_complex & operator+=(const mp_complex & v);
```

14. ```
const mp_complex & operator--(void);
```

15. ```
mp_complex & operator-=(const mp_complex & v);
```

16. ```
mp_complex & operator/=(const mp_complex & v);
```

17. ```
mp_float real(void) const;
```

#### `mp_complex` public static functions

1. ```
static mp_float imag(const mp_complex & z);
```

2. ```
static mp_float real(const mp_complex & z);
```

# Header <boost/multiprecision/mp_float.hpp>

Checks that the user has selected one `mp_float` back-end big-number type by defining `BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_TYPE_xxx`, as a compiler-defined preprocessor macro, where xxx is the backend acronym, for example EFX.
Declares all the very many variants of global operators for mp_float.
Declares constants for use in specialization of `std::numeric_limits<>`.
Provides the specialization of `std::numeric_limits<>` for `mp_float`.
.
```

```
namespace boost {
  namespace multiprecision {
    const mp_float & half(void);
    const mp_float & one(void);
    bool operator!=(const mp_float & u, const mp_float & v);
    bool operator!=(const mp_float & u, const char v);
    bool operator!=(const mp_float & u, const wchar_t v);
    bool operator!=(const mp_float & u, const signed char v);
    bool operator!=(const mp_float & u, const signed short v);
    bool operator!=(const mp_float & u, const int v);
    bool operator!=(const mp_float & u, const signed long v);
    bool operator!=(const mp_float & u, const signed long long v);
    bool operator!=(const mp_float & u, const unsigned char v);
    bool operator!=(const mp_float & u, const unsigned short v);
    bool operator!=(const mp_float & u, const unsigned int v);
    bool operator!=(const mp_float & u, const unsigned long v);
    bool operator!=(const mp_float & u, const unsigned long long v);
    bool operator!=(const mp_float & u, const float v);
    bool operator!=(const mp_float & u, const double v);
    bool operator!=(const mp_float & u, const long double v);
    bool operator!=(const char u, const mp_float & v);
    bool operator!=(const wchar_t u, const mp_float & v);
    bool operator!=(const signed char u, const mp_float & v);
    bool operator!=(const signed short u, const mp_float & v);
    bool operator!=(const int u, const mp_float & v);
    bool operator!=(const signed long u, const mp_float & v);
    bool operator!=(const signed long long u, const mp_float & v);
    bool operator!=(const unsigned char u, const mp_float & v);
    bool operator!=(const unsigned short u, const mp_float & v);
    bool operator!=(const unsigned int u, const mp_float & v);
    bool operator!=(const unsigned long u, const mp_float & v);
    bool operator!=(const unsigned long long u, const mp_float & v);
    bool operator!=(const float u, const mp_float & v);
    bool operator!=(const double u, const mp_float & v);
    bool operator!=(const long double u, const mp_float & v);
    mp_float operator*(const mp_float & u, const mp_float & v);
    mp_float operator*(const mp_float & u, const char n);
    mp_float operator*(const mp_float & u, const signed char n);
    mp_float operator*(const mp_float & u, const signed short n);
    mp_float operator*(const mp_float & u, const int n);
    mp_float operator*(const mp_float & u, const signed long n);
    mp_float operator*(const mp_float & u, const signed long long n);
    mp_float operator*(const mp_float & u, const unsigned char n);
    mp_float operator*(const mp_float & u, const wchar_t n);
    mp_float operator*(const mp_float & u, const unsigned short n);
    mp_float operator*(const mp_float & u, const unsigned int n);
    mp_float operator*(const mp_float & u, const unsigned long n);
    mp_float operator*(const mp_float & u, const unsigned long long n);
    mp_float operator*(const mp_float & u, const float f);
    mp_float operator*(const mp_float & u, const double d);
    mp_float operator*(const mp_float & u, const long double ld);
    mp_float operator*(const char n, const mp_float & u);
    mp_float operator*(const signed char n, const mp_float & u);
    mp_float operator*(const signed short n, const mp_float & u);
    mp_float operator*(const int n, const mp_float & u);
    mp_float operator*(const signed long n, const mp_float & u);
    mp_float operator*(const signed long long n, const mp_float & u);
    mp_float operator*(const unsigned char n, const mp_float & u);
    mp_float operator*(const wchar_t n, const mp_float & u);
    mp_float operator*(const unsigned short n, const mp_float & u);
    mp_float operator*(const unsigned int n, const mp_float & u);
    mp_float operator*(const unsigned long n, const mp_float & u);
    mp_float operator*(const unsigned long long n, const mp_float & u);
```

```
mp_float operator*(const float f, const mp_float & u);
mp_float operator*(const double d, const mp_float & u);
mp_float operator*(const long double ld, const mp_float & u);
mp_float & operator*=(mp_float & u, const char n);
mp_float & operator*=(mp_float & u, const signed char n);
mp_float & operator*=(mp_float & u, const signed short n);
mp_float & operator*=(mp_float & u, const int n);
mp_float & operator*=(mp_float & u, const signed long n);
mp_float & operator*=(mp_float & u, const signed long long n);
mp_float & operator*=(mp_float & u, const unsigned char n);
mp_float & operator*=(mp_float & u, const wchar_t n);
mp_float & operator*=(mp_float & u, const unsigned short n);
mp_float & operator*=(mp_float & u, const unsigned int n);
mp_float & operator*=(mp_float & u, const unsigned long n);
mp_float & operator*=(mp_float & u, const unsigned long long n);
mp_float & operator*=(mp_float & u, const float f);
mp_float & operator*=(mp_float & u, const double d);
mp_float & operator*=(mp_float & u, const long double ld);
mp_float & operator+(mp_float & u);
const mp_float & operator+(const mp_float & u);

// Global add/sub/mul/div of const mp_float& with const mp_float&.
mp_float operator+(const mp_float & u, const mp_float & v);

// Global add/sub/mul/div of const mp_float& with all built-in types.
mp_float operator+(const mp_float & u, const char n);
mp_float operator+(const mp_float & u, const signed char n);
mp_float operator+(const mp_float & u, const signed short n);
mp_float operator+(const mp_float & u, const int n);
mp_float operator+(const mp_float & u, const signed long n);
mp_float operator+(const mp_float & u, const signed long long n);
mp_float operator+(const mp_float & u, const unsigned char n);
mp_float operator+(const mp_float & u, const wchar_t n);
mp_float operator+(const mp_float & u, const unsigned short n);
mp_float operator+(const mp_float & u, const unsigned int n);
mp_float operator+(const mp_float & u, const unsigned long n);
mp_float operator+(const mp_float & u, const unsigned long long n);
mp_float operator+(const mp_float & u, const float f);
mp_float operator+(const mp_float & u, const double d);
mp_float operator+(const mp_float & u, const long double ld);

// Global add/sub/mul/div of all built-in types with const mp_float&.
mp_float operator+(const char n, const mp_float & u);
mp_float operator+(const signed char n, const mp_float & u);
mp_float operator+(const signed short n, const mp_float & u);
mp_float operator+(const int n, const mp_float & u);
mp_float operator+(const signed long n, const mp_float & u);
mp_float operator+(const signed long long n, const mp_float & u);
mp_float operator+(const unsigned char n, const mp_float & u);
mp_float operator+(const wchar_t n, const mp_float & u);
mp_float operator+(const unsigned short n, const mp_float & u);
mp_float operator+(const unsigned int n, const mp_float & u);
mp_float operator+(const unsigned long n, const mp_float & u);
mp_float operator+(const unsigned long long n, const mp_float & u);
mp_float operator+(const float f, const mp_float & u);
mp_float operator+(const double d, const mp_float & u);
mp_float operator+(const long double ld, const mp_float & u);

// Global operators post-increment and post-decrement.
mp_float operator++(mp_float & u, int);

// Global self add/sub/mul/div of mp_float& with all built-in types.
mp_float & operator+=(mp_float & u, const char n);
```

```
mp_float & operator+=(mp_float & u, const signed char n);
mp_float & operator+=(mp_float & u, const signed short n);
mp_float & operator+=(mp_float & u, const int n);
mp_float & operator+=(mp_float & u, const signed long n);
mp_float & operator+=(mp_float & u, const signed long long n);
mp_float & operator+=(mp_float & u, const unsigned char n);
mp_float & operator+=(mp_float & u, const wchar_t n);
mp_float & operator+=(mp_float & u, const unsigned short n);
mp_float & operator+=(mp_float & u, const unsigned int n);
mp_float & operator+=(mp_float & u, const unsigned long n);
mp_float & operator+=(mp_float & u, const unsigned long long n);
mp_float & operator+=(mp_float & u, const float f);
mp_float & operator+=(mp_float & u, const double d);
mp_float & operator+=(mp_float & u, const long double ld);

// Global unary operators of mp_float reference.
mp_float operator-(const mp_float & u);
mp_float operator-(const mp_float & u, const mp_float & v);
mp_float operator-(const mp_float & u, const char n);
mp_float operator-(const mp_float & u, const signed char n);
mp_float operator-(const mp_float & u, const signed short n);
mp_float operator-(const mp_float & u, const int n);
mp_float operator-(const mp_float & u, const signed long n);
mp_float operator-(const mp_float & u, const signed long long n);
mp_float operator-(const mp_float & u, const unsigned char n);
mp_float operator-(const mp_float & u, const wchar_t n);
mp_float operator-(const mp_float & u, const unsigned short n);
mp_float operator-(const mp_float & u, const unsigned int n);
mp_float operator-(const mp_float & u, const unsigned long n);
mp_float operator-(const mp_float & u, const unsigned long long n);
mp_float operator-(const mp_float & u, const float f);
mp_float operator-(const mp_float & u, const double d);
mp_float operator-(const mp_float & u, const long double ld);
mp_float operator-(const char n, const mp_float & u);
mp_float operator-(const signed char n, const mp_float & u);
mp_float operator-(const signed short n, const mp_float & u);
mp_float operator-(const int n, const mp_float & u);
mp_float operator-(const signed long n, const mp_float & u);
mp_float operator-(const signed long long n, const mp_float & u);
mp_float operator-(const unsigned char n, const mp_float & u);
mp_float operator-(const wchar_t n, const mp_float & u);
mp_float operator-(const unsigned short n, const mp_float & u);
mp_float operator-(const unsigned int n, const mp_float & u);
mp_float operator-(const unsigned long n, const mp_float & u);
mp_float operator-(const unsigned long long n, const mp_float & u);
mp_float operator-(const float f, const mp_float & u);
mp_float operator-(const double d, const mp_float & u);
mp_float operator-(const long double ld, const mp_float & u);
mp_float operator--(mp_float & u, int);
mp_float & operator-=(mp_float & u, const signed char n);
mp_float & operator-=(mp_float & u, const signed short n);
mp_float & operator-=(mp_float & u, const int n);
mp_float & operator-=(mp_float & u, const signed long n);
mp_float & operator-=(mp_float & u, const signed long long n);
mp_float & operator-=(mp_float & u, const unsigned char n);
mp_float & operator-=(mp_float & u, const wchar_t n);
mp_float & operator-=(mp_float & u, const unsigned short n);
mp_float & operator-=(mp_float & u, const unsigned int n);
mp_float & operator-=(mp_float & u, const unsigned long n);
mp_float & operator-=(mp_float & u, const unsigned long long n);
mp_float & operator-=(mp_float & u, const float f);
mp_float & operator-=(mp_float & u, const double d);
mp_float & operator-=(mp_float & u, const long double ld);
```

```
mp_float operator/(const mp_float & u, const mp_float & v);
mp_float operator/(const mp_float & u, const char n);
mp_float operator/(const mp_float & u, const signed char n);
mp_float operator/(const mp_float & u, const signed short n);
mp_float operator/(const mp_float & u, const int n);
mp_float operator/(const mp_float & u, const signed long n);
mp_float operator/(const mp_float & u, const signed long long n);
mp_float operator/(const mp_float & u, const unsigned char n);
mp_float operator/(const mp_float & u, const wchar_t n);
mp_float operator/(const mp_float & u, const unsigned short n);
mp_float operator/(const mp_float & u, const unsigned int n);
mp_float operator/(const mp_float & u, const unsigned long n);
mp_float operator/(const mp_float & u, const unsigned long long n);
mp_float operator/(const mp_float & u, const float f);
mp_float operator/(const mp_float & u, const double d);
mp_float operator/(const mp_float & u, const long double ld);
mp_float operator/(const char n, const mp_float & u);
mp_float operator/(const signed char n, const mp_float & u);
mp_float operator/(const signed short n, const mp_float & u);
mp_float operator/(const int n, const mp_float & u);
mp_float operator/(const signed long n, const mp_float & u);
mp_float operator/(const signed long long n, const mp_float & u);
mp_float operator/(const unsigned char n, const mp_float & u);
mp_float operator/(const wchar_t n, const mp_float & u);
mp_float operator/(const unsigned short n, const mp_float & u);
mp_float operator/(const unsigned int n, const mp_float & u);
mp_float operator/(const unsigned long n, const mp_float & u);
mp_float operator/(const unsigned long long n, const mp_float & u);
mp_float operator/(const float f, const mp_float & u);
mp_float operator/(const double d, const mp_float & u);
mp_float operator/(const long double ld, const mp_float & u);
mp_float & operator/=(mp_float & u, const char n);
mp_float & operator/=(mp_float & u, const signed char n);
mp_float & operator/=(mp_float & u, const signed short n);
mp_float & operator/=(mp_float & u, const int n);
mp_float & operator/=(mp_float & u, const signed long n);
mp_float & operator/=(mp_float & u, const signed long long n);
mp_float & operator/=(mp_float & u, const unsigned char n);
mp_float & operator/=(mp_float & u, const wchar_t n);
mp_float & operator/=(mp_float & u, const unsigned short n);
mp_float & operator/=(mp_float & u, const unsigned int n);
mp_float & operator/=(mp_float & u, const unsigned long n);
mp_float & operator/=(mp_float & u, const unsigned long long n);
mp_float & operator/=(mp_float & u, const float f);
mp_float & operator/=(mp_float & u, const double d);
mp_float & operator/=(mp_float & u, const long double ld);

// Global comparison operators of const mp_float& with const mp_float&.
bool operator<(const mp_float & u, const mp_float & v);

// Global comparison operators of const mp_float& with all built-in types.
bool operator<(const mp_float & u, const char v);
bool operator<(const mp_float & u, const wchar_t v);
bool operator<(const mp_float & u, const signed char v);
bool operator<(const mp_float & u, const signed short v);
bool operator<(const mp_float & u, const int v);
bool operator<(const mp_float & u, const signed long v);
bool operator<(const mp_float & u, const signed long long v);
bool operator<(const mp_float & u, const unsigned char v);
bool operator<(const mp_float & u, const unsigned short v);
bool operator<(const mp_float & u, const unsigned int v);
bool operator<(const mp_float & u, const unsigned long v);
bool operator<(const mp_float & u, const unsigned long long v);
```

```
bool operator<(const mp_float & u, const float v);
bool operator<(const mp_float & u, const double v);
bool operator<(const mp_float & u, const long double v);

// Global comparison operators of all built-in types with const mp_float&.
bool operator<(const char u, const mp_float & v);
bool operator<(const wchar_t u, const mp_float & v);
bool operator<(const signed char u, const mp_float & v);
bool operator<(const signed short u, const mp_float & v);
bool operator<(const int u, const mp_float & v);
bool operator<(const signed long u, const mp_float & v);
bool operator<(const signed long long u, const mp_float & v);
bool operator<(const unsigned char u, const mp_float & v);
bool operator<(const unsigned short u, const mp_float & v);
bool operator<(const unsigned int u, const mp_float & v);
bool operator<(const unsigned long u, const mp_float & v);
bool operator<(const unsigned long long u, const mp_float & v);
bool operator<(const float u, const mp_float & v);
bool operator<(const double u, const mp_float & v);
bool operator<(const long double u, const mp_float & v);
bool operator<=(const mp_float & u, const mp_float & v);
bool operator<=(const mp_float & u, const char v);
bool operator<=(const mp_float & u, const wchar_t v);
bool operator<=(const mp_float & u, const signed char v);
bool operator<=(const mp_float & u, const signed short v);
bool operator<=(const mp_float & u, const int v);
bool operator<=(const mp_float & u, const signed long v);
bool operator<=(const mp_float & u, const signed long long v);
bool operator<=(const mp_float & u, const unsigned char v);
bool operator<=(const mp_float & u, const unsigned short v);
bool operator<=(const mp_float & u, const unsigned int v);
bool operator<=(const mp_float & u, const unsigned long v);
bool operator<=(const mp_float & u, const unsigned long long v);
bool operator<=(const mp_float & u, const float v);
bool operator<=(const mp_float & u, const double v);
bool operator<=(const mp_float & u, const long double v);
bool operator<=(const char u, const mp_float & v);
bool operator<=(const wchar_t u, const mp_float & v);
bool operator<=(const signed char u, const mp_float & v);
bool operator<=(const signed short u, const mp_float & v);
bool operator<=(const int u, const mp_float & v);
bool operator<=(const signed long u, const mp_float & v);
bool operator<=(const signed long long u, const mp_float & v);
bool operator<=(const unsigned char u, const mp_float & v);
bool operator<=(const unsigned short u, const mp_float & v);
bool operator<=(const unsigned int u, const mp_float & v);
bool operator<=(const unsigned long u, const mp_float & v);
bool operator<=(const unsigned long long u, const mp_float & v);
bool operator<=(const float u, const mp_float & v);
bool operator<=(const double u, const mp_float & v);
bool operator<=(const long double u, const mp_float & v);
bool operator==(const mp_float & u, const mp_float & v);
bool operator==(const mp_float & u, const char v);
bool operator==(const mp_float & u, const wchar_t v);
bool operator==(const mp_float & u, const signed char v);
bool operator==(const mp_float & u, const signed short v);
bool operator==(const mp_float & u, const int v);
bool operator==(const mp_float & u, const signed long v);
bool operator==(const mp_float & u, const signed long long v);
bool operator==(const mp_float & u, const unsigned char v);
bool operator==(const mp_float & u, const unsigned short v);
bool operator==(const mp_float & u, const unsigned int v);
bool operator==(const mp_float & u, const unsigned long v);
```

```
bool operator==(const mp_float & u, const unsigned long long v);
bool operator==(const mp_float & u, const float v);
bool operator==(const mp_float & u, const double v);
bool operator==(const mp_float & u, const long double v);
bool operator==(const char u, const mp_float & v);
bool operator==(const wchar_t u, const mp_float & v);
bool operator==(const signed char u, const mp_float & v);
bool operator==(const signed short u, const mp_float & v);
bool operator==(const int u, const mp_float & v);
bool operator==(const signed long u, const mp_float & v);
bool operator==(const signed long long u, const mp_float & v);
bool operator==(const unsigned char u, const mp_float & v);
bool operator==(const unsigned short u, const mp_float & v);
bool operator==(const unsigned int u, const mp_float & v);
bool operator==(const unsigned long u, const mp_float & v);
bool operator==(const unsigned long long u, const mp_float & v);
bool operator==(const float u, const mp_float & v);
bool operator==(const double u, const mp_float & v);
bool operator==(const long double u, const mp_float & v);
bool operator>(const mp_float & u, const mp_float & v);
bool operator>(const mp_float & u, const char v);
bool operator>(const mp_float & u, const wchar_t v);
bool operator>(const mp_float & u, const signed char v);
bool operator>(const mp_float & u, const signed short v);
bool operator>(const mp_float & u, const int v);
bool operator>(const mp_float & u, const signed long v);
bool operator>(const mp_float & u, const signed long long v);
bool operator>(const mp_float & u, const unsigned char v);
bool operator>(const mp_float & u, const unsigned short v);
bool operator>(const mp_float & u, const unsigned int v);
bool operator>(const mp_float & u, const unsigned long v);
bool operator>(const mp_float & u, const unsigned long long v);
bool operator>(const mp_float & u, const float v);
bool operator>(const mp_float & u, const double v);
bool operator>(const mp_float & u, const long double v);
bool operator>(const char u, const mp_float & v);
bool operator>(const wchar_t u, const mp_float & v);
bool operator>(const signed char u, const mp_float & v);
bool operator>(const signed short u, const mp_float & v);
bool operator>(const int u, const mp_float & v);
bool operator>(const signed long u, const mp_float & v);
bool operator>(const signed long long u, const mp_float & v);
bool operator>(const unsigned char u, const mp_float & v);
bool operator>(const unsigned short u, const mp_float & v);
bool operator>(const unsigned int u, const mp_float & v);
bool operator>(const unsigned long u, const mp_float & v);
bool operator>(const unsigned long long u, const mp_float & v);
bool operator>(const float u, const mp_float & v);
bool operator>(const double u, const mp_float & v);
bool operator>(const long double u, const mp_float & v);
bool operator>=(const mp_float & u, const mp_float & v);
bool operator>=(const mp_float & u, const char v);
bool operator>=(const mp_float & u, const wchar_t v);
bool operator>=(const mp_float & u, const signed char v);
bool operator>=(const mp_float & u, const signed short v);
bool operator>=(const mp_float & u, const int v);
bool operator>=(const mp_float & u, const signed long v);
bool operator>=(const mp_float & u, const signed long long v);
bool operator>=(const mp_float & u, const unsigned char v);
bool operator>=(const mp_float & u, const unsigned short v);
bool operator>=(const mp_float & u, const unsigned int v);
bool operator>=(const mp_float & u, const unsigned long v);
bool operator>=(const mp_float & u, const unsigned long long v);
```

```
    bool operator>=(const mp_float & u, const float v);
    bool operator>=(const mp_float & u, const double v);
    bool operator>=(const mp_float & u, const long double v);
    bool operator>=(const char u, const mp_float & v);
    bool operator>=(const wchar_t u, const mp_float & v);
    bool operator>=(const signed char u, const mp_float & v);
    bool operator>=(const signed short u, const mp_float & v);
    bool operator>=(const int u, const mp_float & v);
    bool operator>=(const signed long u, const mp_float & v);
    bool operator>=(const signed long long u, const mp_float & v);
    bool operator>=(const unsigned char u, const mp_float & v);
    bool operator>=(const unsigned short u, const mp_float & v);
    bool operator>=(const unsigned int u, const mp_float & v);
    bool operator>=(const unsigned long u, const mp_float & v);
    bool operator>=(const unsigned long long u, const mp_float & v);
    bool operator>=(const float u, const mp_float & v);
    bool operator>=(const double u, const mp_float & v);
    bool operator>=(const long double u, const mp_float & v);
    const mp_float & value_eps(void);
    const mp_float & value_inf(void);
    const mp_float & value_max(void);
    const mp_float & value_min(void);
    const mp_float & value_nan(void);

    // Constant functions for use in the specialization of std::numeric_limits<> for mp_float.
    const mp_float & zero(void);
  }
}namespace std {
  template<> class numeric_limits<boost::multiprecision::mp_float>;
}
```

Class numeric_limits<boost::multiprecision::mp_float>

std::numeric_limits<boost::multiprecision::mp_float>

Synopsis

```
// In header: <boost/multiprecision/mp_float.hpp>


class numeric_limits<boost::multiprecision::mp_float> {
public:

  // public static functions
  static const boost::multiprecision::mp_float & denorm_min(void);
  static const boost::multiprecision::mp_float & epsilon(void);
  static const boost::multiprecision::mp_float & infinity(void);
  static const boost::multiprecision::mp_float & lowest(void);
  static const boost::multiprecision::mp_float &() max(void);
  static const boost::multiprecision::mp_float &() min(void);
  static const boost::multiprecision::mp_float & quiet_NaN(void);
  static const boost::multiprecision::mp_float & round_error(void);
  static const boost::multiprecision::mp_float & signaling_NaN(void);

  // public data members
  static const int digits;
  static const int digits10;
  static const std::float_denorm_style has_denorm;
  static const bool has_denorm_loss;
  static const bool has_infinity;
  static const bool has_quiet_NaN;
  static const bool has_signaling_NaN;
  static const bool is_bounded;
  static const bool is_exact;
  static const bool is_iec559;
  static const bool is_integer;
  static const bool is_modulo;
  static const bool is_signed;
  static const bool is_specialized;
  static const int max_digits10;
  static const boost::int64_t max_exponent;
  static const boost::int64_t max_exponent10;
  static const boost::int64_t min_exponent;
  static const boost::int64_t min_exponent10;
  static const int radix;
  static const std::float_round_style round_style;
  static const bool tinyness_before;
  static const bool traps;
};
```

Description

Provides specialization of `std::numeric_limits` for mp_float.

`numeric_limits` public static functions

1.
```
static const boost::multiprecision::mp_float & denorm_min(void);
```

2.
```
static const boost::multiprecision::mp_float & epsilon(void);
```

3.
```
static const boost::multiprecision::mp_float & infinity(void);
```

4. ```
   static const boost::multiprecision::mp_float & lowest(void);
   ```

5. ```
   static const boost::multiprecision::mp_float &() max(void);
   ```

6. ```
   static const boost::multiprecision::mp_float &() min(void);
   ```

7. ```
   static const boost::multiprecision::mp_float & quiet_NaN(void);
   ```

8. ```
   static const boost::multiprecision::mp_float & round_error(void);
   ```

9. ```
   static const boost::multiprecision::mp_float & signaling_NaN(void);
   ```

Header <boost/multiprecision/mp_float_base.hpp>

Select chosen backend and base class for mp_float.

```
BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_DIGITS10
BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_DIGITS10_LIMIT
```

```
namespace boost {
  namespace multiprecision {
    class mp_float_base;
    std::basic_ostream< char, std::char_traits< char > > &
    operator<<(std::basic_ostream< char, std::char_traits< char > > &,
               const mp_float_base &);
    std::basic_istream< char, std::char_traits< char > > &
    operator>>(std::basic_istream< char, std::char_traits< char > > &,
               mp_float_base &);
    template<const boost::int32_t my_mp_float_digits10>
      boost::int_fast32_t
      template_mp_float_digits10_match_those_of_lib_dll(void);
    template<>
      boost::int_fast32_t
     template_mp_float_digits10_match_those_of_lib_dll<mp_float_base::mp_float_digits10 >(void);
  }
}namespace std {
}
```

Class mp_float_base

boost::multiprecision::mp_float_base

Synopsis

```
// In header: <boost/multiprecision/mp_float_base.hpp>


class mp_float_base {
public:
  // member classes/structs/unions
  template<typename built_in_float_type>
  class built_in_float_parts {
  public:
    // construct/copy/destruct
    built_in_float_parts(const built_in_float_type);
    built_in_float_parts();
    built_in_float_parts(const built_in_float_parts &);
    built_in_float_parts& operator=(const built_in_float_parts &);

    // public member functions
    const int & get_exponent(void) const;
    const unsigned long long & get_mantissa(void) const;

    // private member functions
    void make_parts(const built_in_float_type);
  };
  // construct/copy/destruct
  mp_float_base();
  mp_float_base& operator=(const mp_float &);
  ~mp_float_base();

  // public member functions
  mp_float & add_signed_long_long(const signed long);
  mp_float & add_unsigned_long_long(const unsigned long);
  mp_float & calculate_inv(void);
  mp_float & calculate_sqrt(void);
  boost::int32_t cmp(const mp_float &) const;
  mp_float & div_signed_long_long(const signed long);
  mp_float & div_unsigned_long_long(const unsigned long);
  mp_float extract_decimal_part(void) const;
  double extract_double(void) const;
  mp_float extract_integer_part(void) const;
  long double extract_long_double(void) const;
  void extract_parts(double &, boost::int64_t &) const;
  signed long long extract_signed_long_long(void) const;
  unsigned long long extract_unsigned_long_long(void) const;
  bool has_its_own_acos(void) const;
  bool has_its_own_acosh(void) const;
  bool has_its_own_asin(void) const;
  bool has_its_own_asinh(void) const;
  bool has_its_own_atan(void) const;
  bool has_its_own_atanh(void) const;
  bool has_its_own_cbrt(void) const;
  bool has_its_own_cos(void) const;
  bool has_its_own_cosh(void) const;
  bool has_its_own_cyl_bessel_jn(void) const;
  bool has_its_own_cyl_bessel_yn(void) const;
  bool has_its_own_exp(void) const;
  bool has_its_own_fmod(void) const;
  bool has_its_own_frexp(void) const;
  bool has_its_own_gamma(void) const;
  bool has_its_own_ldexp(void) const;
  bool has_its_own_log(void) const;
  bool has_its_own_riemann_zeta(void) const;
```

```
bool has_its_own_rootn(void) const;
bool has_its_own_sin(void) const;
bool has_its_own_sinh(void) const;
bool has_its_own_tan(void) const;
bool has_its_own_tanh(void) const;
bool isfinite(void) const;
bool isinf(void) const;
bool isint(void) const;
bool isnan(void) const;
bool isneg(void) const;
bool isone(void) const;
bool ispos(void) const;
bool iszero(void) const;
mp_float & mul_signed_long_long(const signed long);
mp_float & mul_unsigned_long_long(const unsigned long);
const mp_float & my_value_inf(void) const;
const mp_float & my_value_max(void) const;
const mp_float & my_value_min(void) const;
const mp_float & my_value_nan(void) const;
mp_float & negate(void);
operator char() const;
operator double() const;
operator float() const;
operator int() const;
operator long double() const;
operator signed char() const;
operator signed long() const;
operator signed long long() const;
operator signed short() const;
operator unsigned char() const;
operator unsigned int() const;
operator unsigned long() const;
operator unsigned long long() const;
operator unsigned short() const;
operator wchar_t() const;
mp_float & operator*=(const mp_float &);
mp_float_base & operator++(void);
mp_float & operator+=(const mp_float &);
mp_float_base & operator--(void);
mp_float & operator-=(const mp_float &);
mp_float & operator/=(const mp_float &);
boost::int64_t order(void) const;
void precision(const boost::int32_t);
bool rd_string(const char *);
mp_float & sub_signed_long_long(const signed long);
mp_float & sub_unsigned_long_long(const unsigned long);
void wr_string(std::string &, std::ostream &) const;

// public static functions
static bool char_is_nonzero_predicate(const char &);
static mp_float my_acos(const mp_float &);
static mp_float my_acosh(const mp_float &);
static mp_float my_asin(const mp_float &);
static mp_float my_asinh(const mp_float &);
static mp_float my_atan(const mp_float &);
static mp_float my_atanh(const mp_float &);
static mp_float my_cbrt(const mp_float &);
static mp_float my_cos(const mp_float &);
static mp_float my_cosh(const mp_float &);
static mp_float my_cyl_bessel_jn(const boost::int32_t, const mp_float &);
static mp_float my_cyl_bessel_yn(const boost::int32_t, const mp_float &);
static mp_float my_exp(const mp_float &);
static mp_float my_fmod(const mp_float &, const mp_float &);
```

```
   static mp_float my_frexp(const mp_float &, int *);
   static mp_float my_gamma(const mp_float &);
   static mp_float my_ldexp(const mp_float &, int);
   static mp_float my_log(const mp_float &);
   static mp_float my_riemann_zeta(const mp_float &);
   static mp_float my_rootn(const mp_float &, const boost::uint32_t);
   static mp_float my_sin(const mp_float &);
   static mp_float my_sinh(const mp_float &);
   static mp_float my_tan(const mp_float &);
   static mp_float my_tanh(const mp_float &);

   // private member functions
   boost::int64_t get_order_exact(void) const;
   boost::int64_t get_order_fast(void) const;
   void get_output_string(std::string &, boost::int64_t &, const std::size_t) const;

   // public data members
   static const boost::int32_t mp_float_digits10;
   static const boost::int32_t mp_float_digits10_extra;
   static const boost::int32_t mp_float_digits10_limit;
   static const boost::int32_t mp_float_digits10_setting;
   static const boost::int32_t mp_float_max_digits10;
};
```

Description

Provides base class and std stream operators for mp_float.

`mp_float_base` public construct/copy/destruct

1. ```
   mp_float_base();
   ```

2. ```
   mp_float_base& operator=(const mp_float &);
   ```

3. ```
   ~mp_float_base();
   ```

`mp_float_base` public member functions

1. ```
   mp_float & add_signed_long_long(const signed long long);
   ```

2. ```
   mp_float & add_unsigned_long_long(const unsigned long long);
   ```

3. ```
   mp_float & calculate_inv(void);
   ```

4. ```
   mp_float & calculate_sqrt(void);
   ```

5. ```
   boost::int32_t cmp(const mp_float &) const;
   ```

6. `mp_float & div_signed_long_long(const signed long long);`

7. `mp_float & div_unsigned_long_long(const unsigned long long);`

8. `mp_float extract_decimal_part(void) const;`

9. `double extract_double(void) const;`

10. `mp_float extract_integer_part(void) const;`

11. `long double extract_long_double(void) const;`

12. `void extract_parts(double &, boost::int64_t &) const;`

13. `signed long long extract_signed_long_long(void) const;`

14. `unsigned long long extract_unsigned_long_long(void) const;`

15. `bool has_its_own_acos(void) const;`

16. `bool has_its_own_acosh(void) const;`

17. `bool has_its_own_asin(void) const;`

18. `bool has_its_own_asinh(void) const;`

19. `bool has_its_own_atan(void) const;`

20. `bool has_its_own_atanh(void) const;`

21. `bool has_its_own_cbrt(void) const;`

```
22. bool has_its_own_cos(void) const;
23. bool has_its_own_cosh(void) const;
24. bool has_its_own_cyl_bessel_jn(void) const;
25. bool has_its_own_cyl_bessel_yn(void) const;
26. bool has_its_own_exp(void) const;
27. bool has_its_own_fmod(void) const;
28. bool has_its_own_frexp(void) const;
29. bool has_its_own_gamma(void) const;
30. bool has_its_own_ldexp(void) const;
31. bool has_its_own_log(void) const;
32. bool has_its_own_riemann_zeta(void) const;
33. bool has_its_own_rootn(void) const;
34. bool has_its_own_sin(void) const;
35. bool has_its_own_sinh(void) const;
36. bool has_its_own_tan(void) const;
37. bool has_its_own_tanh(void) const;
```

```
38. bool isfinite(void) const;

39. bool isinf(void) const;

40. bool isint(void) const;

41. bool isnan(void) const;

42. bool isneg(void) const;

43. bool isone(void) const;

44. bool ispos(void) const;

45. bool iszero(void) const;

46. mp_float & mul_signed_long_long(const signed long long);

47. mp_float & mul_unsigned_long_long(const unsigned long long);

48. const mp_float & my_value_inf(void) const;

49. const mp_float & my_value_max(void) const;

50. const mp_float & my_value_min(void) const;

51. const mp_float & my_value_nan(void) const;

52. mp_float & negate(void);

53. operator char() const;
```

```
54. operator double() const;
55. operator float() const;
56. operator int() const;
57. operator long double() const;
58. operator signed char() const;
59. operator signed long() const;
60. operator signed long long() const;
61. operator signed short() const;
62. operator unsigned char() const;
63. operator unsigned int() const;
64. operator unsigned long() const;
65. operator unsigned long long() const;
66. operator unsigned short() const;
67. operator wchar_t() const;
68. mp_float & operator*=(const mp_float &);
69. mp_float_base & operator++(void);
```

70. ```
mp_float & operator+=(const mp_float &);
```

71. ```
mp_float_base & operator--(void);
```

72. ```
mp_float & operator-=(const mp_float &);
```

73. ```
mp_float & operator/=(const mp_float &);
```

74. ```
boost::int64_t order(void) const;
```

75. ```
void precision(const boost::int32_t);
```

76. ```
bool rd_string(const char * const);
```

77. ```
mp_float & sub_signed_long_long(const signed long long);
```

78. ```
mp_float & sub_unsigned_long_long(const unsigned long long);
```

79. ```
void wr_string(std::string & str, std::ostream & os) const;
```

`mp_float_base` public static functions

1. ```
static bool char_is_nonzero_predicate(const char & c);
```

2. ```
static mp_float my_acos(const mp_float &);
```

3. ```
static mp_float my_acosh(const mp_float &);
```

4. ```
static mp_float my_asin(const mp_float &);
```

5. ```
static mp_float my_asinh(const mp_float &);
```

6. ```
static mp_float my_atan(const mp_float &);
```

```
7.  static mp_float my_atanh(const mp_float &);
8.  static mp_float my_cbrt(const mp_float &);
9.  static mp_float my_cos(const mp_float &);
10. static mp_float my_cosh(const mp_float &);
11. static mp_float my_cyl_bessel_jn(const boost::int32_t, const mp_float &);
12. static mp_float my_cyl_bessel_yn(const boost::int32_t, const mp_float &);
13. static mp_float my_exp(const mp_float &);
14. static mp_float my_fmod(const mp_float &, const mp_float &);
15. static mp_float my_frexp(const mp_float &, int *);
16. static mp_float my_gamma(const mp_float &);
17. static mp_float my_ldexp(const mp_float &, int);
18. static mp_float my_log(const mp_float &);
19. static mp_float my_riemann_zeta(const mp_float &);
20. static mp_float my_rootn(const mp_float &, const boost::uint32_t);
21. static mp_float my_sin(const mp_float &);
22. static mp_float my_sinh(const mp_float &);
```

23.
```
static mp_float my_tan(const mp_float &);
```

24.
```
static mp_float my_tanh(const mp_float &);
```

`mp_float_base` private member functions

1.
```
boost::int64_t get_order_exact(void) const;
```

2.
```
boost::int64_t get_order_fast(void) const;
```

3.
```
void get_output_string(std::string & str, boost::int64_t & my_exp,
                       const std::size_t number_of_digits) const;
```

Class template built_in_float_parts

boost::multiprecision::mp_float_base::built_in_float_parts

Synopsis

```
// In header: <boost/multiprecision/mp_float_base.hpp>


template<typename built_in_float_type>
class built_in_float_parts {
public:
  // construct/copy/destruct
  built_in_float_parts(const built_in_float_type);
  built_in_float_parts();
  built_in_float_parts(const built_in_float_parts &);
  built_in_float_parts& operator=(const built_in_float_parts &);

  // public member functions
  const int & get_exponent(void) const;
  const unsigned long long & get_mantissa(void) const;

  // private member functions
  void make_parts(const built_in_float_type);
};
```

Description

Note

This template can be used with built-in floating-point types like float, double and long double.

Warning

For long double, ensure that the mantissa fits in unsigned long long.

`built_in_float_parts` public construct/copy/destruct

1. ```
   built_in_float_parts(const built_in_float_type f);
   ```

2. ```
   built_in_float_parts();
   ```

3. ```
   built_in_float_parts(const built_in_float_parts &);
   ```

4. ```
   built_in_float_parts& operator=(const built_in_float_parts &);
   ```

`built_in_float_parts` public member functions

1. ```
   const int & get_exponent(void) const;
   ```

2. ```
   const unsigned long long & get_mantissa(void) const;
   ```

`built_in_float_parts` private member functions

1. ```
   void make_parts(const built_in_float_type f);
   ```

Macro BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_DIGITS10

BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_DIGITS10

Synopsis

```
// In header: <boost/multiprecision/mp_float_base.hpp>

BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_DIGITS10
```

Description

Select the number of decimal digits in mp_float by setting the value of BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_DIGITS10. The supported range is 30-300.

Note

Default precision is 50 decimal digits.
```

Warning

This is a **compile-time** constant.
When linking to a library, the linked code must be compiled with the same precision as the library, else the compilation will end with an error message:
The MP float type is undefined! Define the mp_float type! .

Macro BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_DIGITS10_LIMIT

BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_DIGITS10_LIMIT

Synopsis

```
// In header: <boost/multiprecision/mp_float_base.hpp>

BOOST_MULTIPRECISION_BACKEND_MP_FLOAT_DIGITS10_LIMIT
```

Description

A limit on the maximum precision, currently 300 decimal digits.

Header <boost/multiprecision/mp_float_efx.hpp>

```
namespace boost {
  namespace multiprecision {
    class mp_float_efx;
  }
}
```

Class mp_float_efx

boost::multiprecision::mp_float_efx

Synopsis

```
// In header: <boost/multiprecision/mp_float_efx.hpp>


class mp_float_efx : public boost::multiprecision::mp_float_base {
public:
  // construct/copy/destruct
  mp_float_efx();
  mp_float_efx(const char);
  mp_float_efx(const signed char);
  mp_float_efx(const unsigned char);
  mp_float_efx(const wchar_t);
  mp_float_efx(const signed short);
  mp_float_efx(const unsigned short);
  mp_float_efx(const int);
  mp_float_efx(const unsigned int);
  mp_float_efx(const signed long);
  mp_float_efx(const unsigned long);
  mp_float_efx(const signed long long);
  mp_float_efx(const unsigned long long);
  mp_float_efx(const float);
  mp_float_efx(const double);
  mp_float_efx(const long double);
  mp_float_efx(const char *const);
  mp_float_efx(const std::string &);
  mp_float_efx(const mp_float_efx &);
  mp_float_efx(const double, const boost::int64_t);
  mp_float_efx& operator=(const mp_float_efx &);
  ~mp_float_efx();

  // public member functions
  mp_float & add_signed_long_long(const signed long);
  mp_float_efx & add_unsigned_long_long(const unsigned long long);
  mp_float_efx & calculate_inv(void);
  mp_float_efx & calculate_sqrt(void);
  boost::int32_t cmp(const mp_float_efx &) const;
  boost::int32_t cmp(const mp_float &) const;
  mp_float & div_signed_long_long(const signed long);
  mp_float_efx & div_unsigned_long_long(const unsigned long long);
  mp_float_efx extract_decimal_part(void) const;
  double extract_double(void) const;
  mp_float_efx extract_integer_part(void) const;
  long double extract_long_double(void) const;
  void extract_parts(double &, boost::int64_t &) const;
  signed long long extract_signed_long_long(void) const;
  unsigned long long extract_unsigned_long_long(void) const;
  bool has_its_own_acos(void) const;
  bool has_its_own_acosh(void) const;
  bool has_its_own_asin(void) const;
  bool has_its_own_asinh(void) const;
  bool has_its_own_atan(void) const;
  bool has_its_own_atanh(void) const;
  bool has_its_own_cbrt(void) const;
  bool has_its_own_cos(void) const;
  bool has_its_own_cosh(void) const;
  bool has_its_own_cyl_bessel_jn(void) const;
  bool has_its_own_cyl_bessel_yn(void) const;
  bool has_its_own_exp(void) const;
  bool has_its_own_fmod(void) const;
  bool has_its_own_frexp(void) const;
  bool has_its_own_gamma(void) const;
```

```
bool has_its_own_ldexp(void) const;
bool has_its_own_log(void) const;
bool has_its_own_riemann_zeta(void) const;
bool has_its_own_rootn(void) const;
bool has_its_own_sin(void) const;
bool has_its_own_sinh(void) const;
bool has_its_own_tan(void) const;
bool has_its_own_tanh(void) const;
bool isfinite(void) const;
bool isinf(void) const;
bool isint(void) const;
bool isnan(void) const;
bool isneg(void) const;
bool isone(void) const;
bool ispos(void) const;
bool iszero(void) const;
mp_float & mul_signed_long_long(const signed long);
mp_float_efx & mul_unsigned_long_long(const unsigned long long);
const mp_float_efx & my_value_inf(void) const;
const mp_float & my_value_max(void) const;
const mp_float & my_value_min(void) const;
const mp_float_efx & my_value_nan(void) const;
mp_float_efx & negate(void);
operator char() const;
operator double() const;
operator float() const;
operator int() const;
operator long double() const;
operator signed char() const;
operator signed long() const;
operator signed long long() const;
operator signed short() const;
operator unsigned char() const;
operator unsigned int() const;
operator unsigned long() const;
operator unsigned long long() const;
operator unsigned short() const;
operator wchar_t() const;
mp_float & operator*=(const mp_float &);
mp_float_efx & operator*=(const mp_float_efx &);
mp_float_efx & operator++(void);
mp_float & operator+=(const mp_float &);
mp_float_efx & operator+=(const mp_float_efx &);
mp_float_efx & operator--(void);
mp_float & operator-=(const mp_float &);
mp_float_efx & operator-=(const mp_float_efx &);
mp_float & operator/=(const mp_float &);
mp_float_efx & operator/=(const mp_float_efx &);
boost::int64_t order(void) const;
void precision(const boost::int32_t);
mp_float & sub_signed_long_long(const signed long);
mp_float_efx & sub_unsigned_long_long(const unsigned long long);
void wr_string(std::string &, std::ostream &) const;

// private static functions
static bool data_elem_is_non_nine_predicate(const boost::uint32_t &);
static bool data_elem_is_non_zero_predicate(const boost::uint32_t &);
static boost::uint32_t
div_loop_n(boost::uint32_t *const, boost::uint32_t, const boost::int32_t);
static boost::uint32_t
mul_loop_n(boost::uint32_t *const, boost::uint32_t, const boost::int32_t);
static void mul_loop_uv(const boost::uint32_t *const,
                        const boost::uint32_t *const,
```

```
                        boost::uint32_t *const, const boost::int32_t);

  // private member functions
  boost::int32_t cmp_data(const array_type &) const;
  void from_unsigned_long(const unsigned long);
  void from_unsigned_long_long(const unsigned long long);
  boost::int64_t get_order_exact(void) const;
  boost::int64_t get_order_fast(void) const;
  void get_output_string(std::string &, boost::int64_t &, const std::size_t) const;
  bool rd_string(const char *const);

  // public static functions
  static bool char_is_nonzero_predicate(const char &);
  static mp_float my_acos(const mp_float &);
  static mp_float my_acosh(const mp_float &);
  static mp_float my_asin(const mp_float &);
  static mp_float my_asinh(const mp_float &);
  static mp_float my_atan(const mp_float &);
  static mp_float my_atanh(const mp_float &);
  static mp_float my_cbrt(const mp_float &);
  static mp_float my_cos(const mp_float &);
  static mp_float my_cosh(const mp_float &);
  static mp_float my_cyl_bessel_jn(const boost::int32_t, const mp_float &);
  static mp_float my_cyl_bessel_yn(const boost::int32_t, const mp_float &);
  static mp_float my_exp(const mp_float &);
  static mp_float my_fmod(const mp_float &, const mp_float &);
  static mp_float my_frexp(const mp_float &, int *);
  static mp_float my_gamma(const mp_float &);
  static mp_float my_ldexp(const mp_float &, int);
  static mp_float my_log(const mp_float &);
  static mp_float my_riemann_zeta(const mp_float &);
  static mp_float my_rootn(const mp_float &, const boost::uint32_t);
  static mp_float my_sin(const mp_float &);
  static mp_float my_sinh(const mp_float &);
  static mp_float my_tan(const mp_float &);
  static mp_float my_tanh(const mp_float &);

  // public data members
  static const boost::int32_t mp_elem_digits10;
  static const boost::int32_t mp_float_digits;
  static const boost::int64_t mp_float_max_exp;
  static const boost::int64_t mp_float_max_exp10;
  static const boost::int64_t mp_float_min_exp;
  static const boost::int64_t mp_float_min_exp10;
  static const boost::int32_t mp_radix;
};
```

Description

`mp_float_efx` public construct/copy/destruct

1. ```
   mp_float_efx();
   ```

2. ```
   mp_float_efx(const char n);
   ```

3. ```
   mp_float_efx(const signed char n);
   ```
   ```

4. `mp_float_efx(const unsigned char n);`
5. `mp_float_efx(const wchar_t n);`
6. `mp_float_efx(const signed short n);`
7. `mp_float_efx(const unsigned short n);`
8. `mp_float_efx(const int n);`
9. `mp_float_efx(const unsigned int n);`
10. `mp_float_efx(const signed long n);`
11. `mp_float_efx(const unsigned long n);`
12. `mp_float_efx(const signed long long n);`
13. `mp_float_efx(const unsigned long long n);`
14. `mp_float_efx(const float f);`
15. `mp_float_efx(const double d);`
16. `mp_float_efx(const long double ld);`
17. `mp_float_efx(const char *const s);`
18. `mp_float_efx(const std::string & str);`
19. `mp_float_efx(const mp_float_efx & f);`

20. `mp_float_efx(const double mantissa, const boost::int64_t exponent);`

21. `mp_float_efx& operator=(const mp_float_efx & v);`

22. `~mp_float_efx();`

`mp_float_efx` public member functions

1. `mp_float & add_signed_long_long(const signed long long);`

2. `mp_float_efx & add_unsigned_long_long(const unsigned long long n);`

3. `mp_float_efx & calculate_inv(void);`

4. `mp_float_efx & calculate_sqrt(void);`

5. `boost::int32_t cmp(const mp_float_efx & v) const;`

6. `boost::int32_t cmp(const mp_float &) const;`

7. `mp_float & div_signed_long_long(const signed long long);`

8. `mp_float_efx & div_unsigned_long_long(const unsigned long long n);`

9. `mp_float_efx extract_decimal_part(void) const;`

10. `double extract_double(void) const;`

11. `mp_float_efx extract_integer_part(void) const;`

12. `long double extract_long_double(void) const;`

13. `void extract_parts(double & mantissa, boost::int64_t & exponent) const;`

```
14. signed long long extract_signed_long_long(void) const;

15. unsigned long long extract_unsigned_long_long(void) const;

16. bool has_its_own_acos(void) const;

17. bool has_its_own_acosh(void) const;

18. bool has_its_own_asin(void) const;

19. bool has_its_own_asinh(void) const;

20. bool has_its_own_atan(void) const;

21. bool has_its_own_atanh(void) const;

22. bool has_its_own_cbrt(void) const;

23. bool has_its_own_cos(void) const;

24. bool has_its_own_cosh(void) const;

25. bool has_its_own_cyl_bessel_jn(void) const;

26. bool has_its_own_cyl_bessel_yn(void) const;

27. bool has_its_own_exp(void) const;

28. bool has_its_own_fmod(void) const;

29. bool has_its_own_frexp(void) const;
```

```
30. bool has_its_own_gamma(void) const;
31. bool has_its_own_ldexp(void) const;
32. bool has_its_own_log(void) const;
33. bool has_its_own_riemann_zeta(void) const;
34. bool has_its_own_rootn(void) const;
35. bool has_its_own_sin(void) const;
36. bool has_its_own_sinh(void) const;
37. bool has_its_own_tan(void) const;
38. bool has_its_own_tanh(void) const;
39. bool isfinite(void) const;
40. bool isinf(void) const;
41. bool isint(void) const;
42. bool isnan(void) const;
43. bool isneg(void) const;
44. bool isone(void) const;
45. bool ispos(void) const;
```

```
46. bool iszero(void) const;

47. mp_float & mul_signed_long_long(const signed long long);

48. mp_float_efx & mul_unsigned_long_long(const unsigned long long n);

49. const mp_float_efx & my_value_inf(void) const;

50. const mp_float & my_value_max(void) const;

51. const mp_float & my_value_min(void) const;

52. const mp_float_efx & my_value_nan(void) const;

53. mp_float_efx & negate(void);

54. operator char() const;

55. operator double() const;

56. operator float() const;

57. operator int() const;

58. operator long double() const;

59. operator signed char() const;

60. operator signed long() const;

61. operator signed long long() const;
```

```
62. operator signed short() const;

63. operator unsigned char() const;

64. operator unsigned int() const;

65. operator unsigned long() const;

66. operator unsigned long long() const;

67. operator unsigned short() const;

68. operator wchar_t() const;

69. mp_float & operator*=(const mp_float &);

70. mp_float_efx & operator*=(const mp_float_efx & v);

71. mp_float_efx & operator++(void);

72. mp_float & operator+=(const mp_float &);

73. mp_float_efx & operator+=(const mp_float_efx & v);

74. mp_float_efx & operator--(void);

75. mp_float & operator-=(const mp_float &);

76. mp_float_efx & operator-=(const mp_float_efx & v);

77. mp_float & operator/=(const mp_float &);
```

78.
```
mp_float_efx & operator/=(const mp_float_efx & v);
```

79.
```
boost::int64_t order(void) const;
```

80.
```
void precision(const boost::int32_t prec_digits);
```

81.
```
mp_float & sub_signed_long_long(const signed long long n);
```

82.
```
mp_float_efx & sub_unsigned_long_long(const unsigned long long n);
```

83.
```
void wr_string(std::string & str, std::ostream & os) const;
```

`mp_float_efx` private static functions

1.
```
static bool data_elem_is_non_nine_predicate(const boost::uint32_t & d);
```

2.
```
static bool data_elem_is_non_zero_predicate(const boost::uint32_t & d);
```

3.
```
static boost::uint32_t
div_loop_n(boost::uint32_t *const u, boost::uint32_t n,
           const boost::int32_t p);
```

4.
```
static boost::uint32_t
mul_loop_n(boost::uint32_t *const u, boost::uint32_t n,
           const boost::int32_t p);
```

5.
```
static void mul_loop_uv(const boost::uint32_t *const u,
                        const boost::uint32_t *const v,
                        boost::uint32_t *const w, const boost::int32_t p);
```

`mp_float_efx` private member functions

1.
```
boost::int32_t cmp_data(const array_type & vd) const;
```

2.
```
void from_unsigned_long(const unsigned long u);
```

3.
```
void from_unsigned_long_long(const unsigned long long u);
```

4.
```
boost::int64_t get_order_exact(void) const;
```

5.
```
boost::int64_t get_order_fast(void) const;
```

6.
```
void get_output_string(std::string & str, boost::int64_t & my_exp,
                       const std::size_t number_of_digits) const;
```

7.
```
bool rd_string(const char *const s);
```

`mp_float_efx` public static functions

1.
```
static bool char_is_nonzero_predicate(const char & c);
```

2.
```
static mp_float my_acos(const mp_float &);
```

3.
```
static mp_float my_acosh(const mp_float &);
```

4.
```
static mp_float my_asin(const mp_float &);
```

5.
```
static mp_float my_asinh(const mp_float &);
```

6.
```
static mp_float my_atan(const mp_float &);
```

7.
```
static mp_float my_atanh(const mp_float &);
```

8.
```
static mp_float my_cbrt(const mp_float &);
```

9.
```
static mp_float my_cos(const mp_float &);
```

10.
```
static mp_float my_cosh(const mp_float &);
```

11.
```
static mp_float my_cyl_bessel_jn(const boost::int32_t, const mp_float &);
```

12.
```
static mp_float my_cyl_bessel_yn(const boost::int32_t, const mp_float &);
```

```
13. static mp_float my_exp(const mp_float &);
14. static mp_float my_fmod(const mp_float &, const mp_float &);
15. static mp_float my_frexp(const mp_float &, int *);
16. static mp_float my_gamma(const mp_float &);
17. static mp_float my_ldexp(const mp_float &, int);
18. static mp_float my_log(const mp_float &);
19. static mp_float my_riemann_zeta(const mp_float &);
20. static mp_float my_rootn(const mp_float &, const boost::uint32_t);
21. static mp_float my_sin(const mp_float &);
22. static mp_float my_sinh(const mp_float &);
23. static mp_float my_tan(const mp_float &);
24. static mp_float my_tanh(const mp_float &);
```

Header <boost/multiprecision/mp_float_functions.hpp>

```
namespace boost {
  namespace multiprecision {
    boost::multiprecision::mp_float
    abs(const boost::multiprecision::mp_float & x);
    boost::multiprecision::mp_float
    abs(const boost::multiprecision::mp_complex & z);
    boost::multiprecision::mp_float
    acos(const boost::multiprecision::mp_float & x);
    boost::multiprecision::mp_complex
    acos(const boost::multiprecision::mp_complex & z);
    boost::multiprecision::mp_float
    acosh(const boost::multiprecision::mp_float & x);
    boost::multiprecision::mp_complex
    acosh(const boost::multiprecision::mp_complex & z);
    boost::multiprecision::mp_float
    arg(const boost::multiprecision::mp_complex & z);
    boost::multiprecision::mp_float
    asin(const boost::multiprecision::mp_float & x);
    boost::multiprecision::mp_complex
    asin(const boost::multiprecision::mp_complex & z);
    boost::multiprecision::mp_float
    asinh(const boost::multiprecision::mp_float & x);
    boost::multiprecision::mp_complex
    asinh(const boost::multiprecision::mp_complex & z);
    boost::multiprecision::mp_float
    atan(const boost::multiprecision::mp_float & x);
    boost::multiprecision::mp_complex
    atan(const boost::multiprecision::mp_complex & z);
    boost::multiprecision::mp_float
    atan2(const boost::multiprecision::mp_float & y,
          const boost::multiprecision::mp_float & x);
    boost::multiprecision::mp_float
    atanh(const boost::multiprecision::mp_float & x);
    boost::multiprecision::mp_complex
    atanh(const boost::multiprecision::mp_complex & z);
    boost::multiprecision::mp_float bernoulli(const boost::uint32_t n);
    void bernoulli_table(std::vector< boost::multiprecision::mp_float > & bn,
                         const boost::uint32_t n);
    const boost::multiprecision::mp_float & billion(void);
    boost::multiprecision::mp_float
    binomial(const boost::uint32_t n, const boost::uint32_t k);
    boost::multiprecision::mp_float
    binomial(const boost::uint32_t n,
             const boost::multiprecision::mp_float & y);
    boost::multiprecision::mp_float
    binomial(const boost::multiprecision::mp_float & x,
             const boost::uint32_t k);
    boost::multiprecision::mp_float
    binomial(const boost::multiprecision::mp_float & x,
             const boost::multiprecision::mp_float & y);
    const boost::multiprecision::mp_float & catalan(void);
    boost::multiprecision::mp_float
    cbrt(const boost::multiprecision::mp_float & x);
    boost::multiprecision::mp_float
    ceil(const boost::multiprecision::mp_float & x);
    boost::multiprecision::mp_complex
    conj(const boost::multiprecision::mp_complex & z);
    boost::multiprecision::mp_float
    cos(const boost::multiprecision::mp_float & x);
    boost::multiprecision::mp_complex
```

```
cos(const boost::multiprecision::mp_complex & z);
boost::multiprecision::mp_float
cosh(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_complex
cosh(const boost::multiprecision::mp_complex & z);
boost::multiprecision::mp_float
cot(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_complex
cot(const boost::multiprecision::mp_complex & z);
boost::multiprecision::mp_float
csc(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_complex
csc(const boost::multiprecision::mp_complex & z);
boost::multiprecision::mp_float
decimal_part(const boost::multiprecision::mp_float & x);
const boost::multiprecision::mp_float & degree(void);
const boost::multiprecision::mp_float & double_max(void);
const boost::multiprecision::mp_float & double_min(void);
const boost::multiprecision::mp_float & eight(void);
const boost::multiprecision::mp_float & eighth(void);
const boost::multiprecision::mp_float & euler_gamma(void);
boost::multiprecision::mp_float
exp(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_complex
exp(const boost::multiprecision::mp_complex & z);
const boost::multiprecision::mp_float & exp1(void);
const boost::multiprecision::mp_float & extreme_value_skewness(void);
boost::multiprecision::mp_float
fabs(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_float factorial(const boost::uint32_t n);
boost::multiprecision::mp_float factorial2(const boost::int32_t n);
const boost::multiprecision::mp_float & fifth(void);
const boost::multiprecision::mp_float & fifty(void);
const boost::multiprecision::mp_float & fifty_k(void);
const boost::multiprecision::mp_float & five(void);
const boost::multiprecision::mp_float & five_hundred(void);
const boost::multiprecision::mp_float & five_k(void);
boost::multiprecision::mp_float
floor(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_float
fmod(const boost::multiprecision::mp_float & v1,
     const boost::multiprecision::mp_float & v2);
const boost::multiprecision::mp_float & forty(void);
const boost::multiprecision::mp_float & forty_k(void);
const boost::multiprecision::mp_float & four(void);
const boost::multiprecision::mp_float & four_hundred(void);
const boost::multiprecision::mp_float & four_k(void);
const boost::multiprecision::mp_float & four_third(void);
boost::multiprecision::mp_float
frexp(const boost::multiprecision::mp_float & v, int * expon);
boost::multiprecision::mp_float
gamma(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_complex
gamma(const boost::multiprecision::mp_complex & z);
boost::multiprecision::mp_float
gamma_near_n(const boost::int32_t n,
             const boost::multiprecision::mp_float & x);
const boost::multiprecision::mp_float & glaisher(void);
const boost::multiprecision::mp_float & golden_ratio(void);
const boost::multiprecision::mp_float & googol(void);
const boost::multiprecision::mp_float & hundred(void);
const boost::multiprecision::mp_float & hundred_k(void);
const boost::multiprecision::mp_float & hundred_M(void);
```

```
boost::multiprecision::mp_float
hyp0F0(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_float
hyp0F1(const boost::multiprecision::mp_float & b,
       const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_float
hyp1F0(const boost::multiprecision::mp_float & a,
       const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_float
hyp1F1(const boost::multiprecision::mp_float & a,
       const boost::multiprecision::mp_float & b,
       const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_float
hyp2F0(const boost::multiprecision::mp_float & a,
       const boost::multiprecision::mp_float & b,
       const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_float
hyp2F1(const boost::multiprecision::mp_float & a,
       const boost::multiprecision::mp_float & b,
       const boost::multiprecision::mp_float & c,
       const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_float
hypPFQ(const std::deque< boost::multiprecision::mp_float > & a,
       const std::deque< boost::multiprecision::mp_float > & b,
       const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_float
imag(const boost::multiprecision::mp_float &);
boost::multiprecision::mp_float
imag(const boost::multiprecision::mp_complex & z);
const boost::multiprecision::mp_float & int32_max(void);
const boost::multiprecision::mp_float & int32_min(void);
const boost::multiprecision::mp_float & int64_max(void);
const boost::multiprecision::mp_float & int64_min(void);
boost::multiprecision::mp_float
integer_part(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_float
inv(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_complex
inv(const boost::multiprecision::mp_complex & z);
bool isfinite(const double x);
bool isfinite(const boost::multiprecision::mp_float & x);
bool isfinite(const boost::multiprecision::mp_complex & z);
bool isinf(const double x);
bool isinf(const boost::multiprecision::mp_float & x);
bool isinf(const boost::multiprecision::mp_complex & z);
bool isint(const double x);
bool isint(const boost::multiprecision::mp_float & x);
bool isint(const boost::multiprecision::mp_complex & z);
bool isnan(const double x);
bool isnan(const boost::multiprecision::mp_float & x);
bool isnan(const boost::multiprecision::mp_complex & z);
bool isneg(const double x);
bool isneg(const boost::multiprecision::mp_float & x);
bool isneg(const boost::multiprecision::mp_complex & z);
bool isone(const double x);
bool isone(const boost::multiprecision::mp_float & x);
bool isone(const boost::multiprecision::mp_complex & z);
bool ispos(const double x);
bool ispos(const boost::multiprecision::mp_float & x);
bool ispos(const boost::multiprecision::mp_complex & z);
bool iszero(const double x);
bool iszero(const boost::multiprecision::mp_float & x);
bool iszero(const boost::multiprecision::mp_complex & z);
```

```
boost::multiprecision::mp_complex
iz(const boost::multiprecision::mp_complex & z);
const boost::multiprecision::mp_float & khinchin(void);
bool large_arg(const double x);
bool large_arg(const boost::multiprecision::mp_float & x);
bool large_arg(const boost::multiprecision::mp_complex & z);
boost::multiprecision::mp_float
ldexp(const boost::multiprecision::mp_float & v, int e);
const boost::multiprecision::mp_float & ln10(void);
const boost::multiprecision::mp_float & ln2(void);
const boost::multiprecision::mp_float & ln3(void);
boost::multiprecision::mp_float
log(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_complex
log(const boost::multiprecision::mp_complex & z);
boost::multiprecision::mp_float
log10(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_complex
log10(const boost::multiprecision::mp_complex & z);
const boost::multiprecision::mp_float & log10_2(void);
boost::multiprecision::mp_float
log1p(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_float
log1p1m2(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_float
loga(const boost::multiprecision::mp_float & a,
     const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_complex
loga(const boost::multiprecision::mp_complex & a,
     const boost::multiprecision::mp_complex & z);
const boost::multiprecision::mp_float & long_double_max(void);
const boost::multiprecision::mp_float & long_double_min(void);
boost::int32_t max_iteration(void);
const boost::multiprecision::mp_float & million(void);
bool near_int(const double x);
bool near_int(const boost::multiprecision::mp_float & x);
bool near_int(const boost::multiprecision::mp_complex & z);
bool near_one(const double x);
bool near_one(const boost::multiprecision::mp_float & x);
bool near_one(const boost::multiprecision::mp_complex & z);
const boost::multiprecision::mp_float & nine(void);
boost::multiprecision::mp_float
norm(const boost::multiprecision::mp_complex & z);
const boost::multiprecision::mp_float & one_minus(void);
boost::int64_t order_of(const double x);
boost::int64_t order_of(const boost::multiprecision::mp_float & x);
const boost::multiprecision::mp_float & pi(void);
const boost::multiprecision::mp_float & pi_half(void);
const boost::multiprecision::mp_float & pi_quarter(void);
const boost::multiprecision::mp_float & pi_squared(void);
boost::multiprecision::mp_float
pochhammer(const boost::multiprecision::mp_float & x,
           const boost::uint32_t n);
boost::multiprecision::mp_float
pochhammer(const boost::multiprecision::mp_float & x,
           const boost::multiprecision::mp_float & a);
boost::multiprecision::mp_complex
pochhammer(const boost::multiprecision::mp_complex & z,
           const boost::uint32_t n);
boost::multiprecision::mp_complex
pochhammer(const boost::multiprecision::mp_complex & z,
           const boost::multiprecision::mp_complex & a);
boost::multiprecision::mp_complex
```

```
polar(const boost::multiprecision::mp_float & mod,
      const boost::multiprecision::mp_float & arg);
boost::multiprecision::mp_float
pow(const boost::multiprecision::mp_float & x,
    const boost::multiprecision::mp_float & a);
boost::multiprecision::mp_complex
pow(const boost::multiprecision::mp_complex & z,
    const boost::multiprecision::mp_complex & a);
boost::multiprecision::mp_float pow2(const boost::int64_t p);
boost::multiprecision::mp_float
pown(const boost::multiprecision::mp_float & x, const boost::int64_t p);
boost::multiprecision::mp_complex
pown(const boost::multiprecision::mp_complex & z, const boost::int64_t p);
void prime(const boost::uint32_t n,
           std::deque< boost::uint32_t > & primes);
const boost::multiprecision::mp_float & quarter(void);
const boost::multiprecision::mp_float & rayleigh_kurtosis(void);
const boost::multiprecision::mp_float & rayleigh_kurtosis_excess(void);
const boost::multiprecision::mp_float & rayleigh_skewness(void);
boost::multiprecision::mp_float
real(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_float
real(const boost::multiprecision::mp_complex & z);
boost::multiprecision::mp_float riemann_zeta(const boost::int32_t n);
boost::multiprecision::mp_float
riemann_zeta(const boost::multiprecision::mp_float & s);
boost::multiprecision::mp_complex
riemann_zeta(const boost::multiprecision::mp_complex & s);
boost::multiprecision::mp_float
rootn(const boost::multiprecision::mp_float & x, const boost::int32_t p);
boost::multiprecision::mp_complex
rootn(const boost::multiprecision::mp_complex & z, const boost::int32_t p);
boost::multiprecision::mp_float
sec(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_complex
sec(const boost::multiprecision::mp_complex & z);
const boost::multiprecision::mp_float & seven(void);
boost::int32_t sgn(const boost::multiprecision::mp_float & x);
const boost::multiprecision::mp_float & signed_long_long_max(void);
const boost::multiprecision::mp_float & signed_long_long_min(void);
boost::multiprecision::mp_float
sin(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_complex
sin(const boost::multiprecision::mp_complex & z);
void sincos(const boost::multiprecision::mp_float & x,
            boost::multiprecision::mp_float *const p_sin,
            boost::multiprecision::mp_float *const p_cos);
void sincos(const boost::multiprecision::mp_complex & z,
            boost::multiprecision::mp_complex *const p_sin,
            boost::multiprecision::mp_complex *const p_cos);
boost::multiprecision::mp_float
sinh(const boost::multiprecision::mp_float & x);
boost::multiprecision::mp_complex
sinh(const boost::multiprecision::mp_complex & z);
void sinhcosh(const boost::multiprecision::mp_float & x,
              boost::multiprecision::mp_float *const p_sin,
              boost::multiprecision::mp_float *const p_cos);
void sinhcosh(const boost::multiprecision::mp_complex & z,
              boost::multiprecision::mp_complex *const p_sinh,
              boost::multiprecision::mp_complex *const p_cosh);
const boost::multiprecision::mp_float & six(void);
const boost::multiprecision::mp_float & sixteenth(void);
bool small_arg(const double x);
```

```
    bool small_arg(const boost::multiprecision::mp_float & x);
    bool small_arg(const boost::multiprecision::mp_complex & z);
    boost::multiprecision::mp_float
    sqrt(const boost::multiprecision::mp_float & x);
    boost::multiprecision::mp_complex
    sqrt(const boost::multiprecision::mp_complex & z);
    const boost::multiprecision::mp_float & sqrt2(void);
    const boost::multiprecision::mp_float & sqrt3(void);
    const boost::multiprecision::mp_float & sqrt_pi(void);
    boost::multiprecision::mp_float
    tan(const boost::multiprecision::mp_float & x);
    boost::multiprecision::mp_complex
    tan(const boost::multiprecision::mp_complex & z);
    boost::multiprecision::mp_float
    tanh(const boost::multiprecision::mp_float & x);
    boost::multiprecision::mp_complex
    tanh(const boost::multiprecision::mp_complex & z);
    const boost::multiprecision::mp_float & ten(void);
    const boost::multiprecision::mp_float & ten_k(void);
    const boost::multiprecision::mp_float & ten_M(void);
    const boost::multiprecision::mp_float & tenth(void);
    const boost::multiprecision::mp_float & third(void);
    const boost::multiprecision::mp_float & thirty(void);
    const boost::multiprecision::mp_float & thirty_k(void);
    const boost::multiprecision::mp_float & thousand(void);
    const boost::multiprecision::mp_float & three(void);
    const boost::multiprecision::mp_float & three_half(void);
    const boost::multiprecision::mp_float & three_hundred(void);
    const boost::multiprecision::mp_float & three_k(void);
    double to_double(const double & x);
    double to_double(const boost::multiprecision::mp_float & x);
    double to_double(const boost::multiprecision::mp_complex & z);
    boost::int32_t to_int32(const double x);
    boost::int32_t to_int32(const boost::multiprecision::mp_float & x);
    boost::int32_t to_int32(const boost::multiprecision::mp_complex & z);
    boost::int64_t to_int64(const double x);
    boost::int64_t to_int64(const boost::multiprecision::mp_float & x);
    boost::int64_t to_int64(const boost::multiprecision::mp_complex & z);
    void to_parts(const boost::multiprecision::mp_float & x,
                  double & mantissa, boost::int64_t & exponent);
    boost::int64_t tol(void);
    const boost::multiprecision::mp_float & trillion(void);
    const boost::multiprecision::mp_float & twenty(void);
    const boost::multiprecision::mp_float & twenty_k(void);
    const boost::multiprecision::mp_float & two(void);
    const boost::multiprecision::mp_float & two_hundred(void);
    const boost::multiprecision::mp_float & two_k(void);
    const boost::multiprecision::mp_float & two_pi(void);
    const boost::multiprecision::mp_float & two_third(void);
    const boost::multiprecision::mp_float & unsigned_long_long_max(void);
  }
}
```

Header <boost/multiprecision/mp_float_gmp.hpp>

```
struct struct__mpf_struct;namespace boost {
  namespace multiprecision {
    class mp_float_gmp;
  }
}

typedef struct struct__mpf_struct __mpf_struct;
typedef long int mp_exp_t;
typedef unsigned long int mp_limb_t;
typedef long int mp_size_t;
typedef __mpf_struct * mpf_ptr;
typedef const __mpf_struct * mpf_srcptr;
typedef __mpf_struct mpf_t;
```

Struct struct__mpf_struct

struct__mpf_struct

Synopsis

```
// In header: <boost/multiprecision/mp_float_gmp.hpp>


struct struct__mpf_struct {

  // public data members
  mp_limb_t * _mp_d;
  mp_exp_t _mp_exp;
  int _mp_prec;
  int _mp_size;
};
```

Class mp_float_gmp

boost::multiprecision::mp_float_gmp

Synopsis

```
// In header: <boost/multiprecision/mp_float_gmp.hpp>


class mp_float_gmp : public boost::multiprecision::mp_float_base {
public:
  // construct/copy/destruct
  mp_float_gmp();
  mp_float_gmp(const char);
  mp_float_gmp(const signed char);
  mp_float_gmp(const unsigned char);
  mp_float_gmp(const wchar_t);
  mp_float_gmp(const signed short);
  mp_float_gmp(const unsigned short);
  mp_float_gmp(const int);
  mp_float_gmp(const unsigned int);
  mp_float_gmp(const signed long);
  mp_float_gmp(const unsigned long);
  mp_float_gmp(const signed long long);
  mp_float_gmp(const unsigned long long);
  mp_float_gmp(const float);
  mp_float_gmp(const double);
  mp_float_gmp(const long double);
  mp_float_gmp(const char *const);
  mp_float_gmp(const std::string &);
  mp_float_gmp(const mp_float_gmp &);
  mp_float_gmp(const double, const boost::int64_t);
  explicit mp_float_gmp(const ::mpf_t &);
  mp_float_gmp& operator=(const mp_float_gmp &);
  ~mp_float_gmp();

  // public member functions
  mp_float & add_signed_long_long(const signed long);
  mp_float_gmp & add_unsigned_long_long(const unsigned long long);
  mp_float_gmp & calculate_inv(void);
  mp_float_gmp & calculate_sqrt(void);
  boost::int32_t cmp(const mp_float_gmp &) const;
  boost::int32_t cmp(const mp_float &) const;
  mp_float & div_signed_long_long(const signed long);
  mp_float_gmp & div_unsigned_long_long(const unsigned long long);
  mp_float_gmp extract_decimal_part(void) const;
  double extract_double(void) const;
  mp_float_gmp extract_integer_part(void) const;
  long double extract_long_double(void) const;
  void extract_parts(double &, boost::int64_t &) const;
  signed long long extract_signed_long_long(void) const;
  unsigned long long extract_unsigned_long_long(void) const;
  bool has_its_own_acos(void) const;
  bool has_its_own_acosh(void) const;
  bool has_its_own_asin(void) const;
  bool has_its_own_asinh(void) const;
  bool has_its_own_atan(void) const;
  bool has_its_own_atanh(void) const;
  bool has_its_own_cbrt(void) const;
  bool has_its_own_cos(void) const;
  bool has_its_own_cosh(void) const;
  bool has_its_own_cyl_bessel_jn(void) const;
  bool has_its_own_cyl_bessel_yn(void) const;
  bool has_its_own_exp(void) const;
  bool has_its_own_fmod(void) const;
  bool has_its_own_frexp(void) const;
```

```
bool has_its_own_gamma(void) const;
bool has_its_own_ldexp(void) const;
bool has_its_own_log(void) const;
bool has_its_own_riemann_zeta(void) const;
bool has_its_own_rootn(void) const;
bool has_its_own_sin(void) const;
bool has_its_own_sinh(void) const;
bool has_its_own_tan(void) const;
bool has_its_own_tanh(void) const;
bool isfinite(void) const;
bool isinf(void) const;
bool isint(void) const;
bool isnan(void) const;
bool isneg(void) const;
bool isone(void) const;
bool ispos(void) const;
bool iszero(void) const;
mp_float & mul_signed_long_long(const signed long);
mp_float_gmp & mul_unsigned_long_long(const unsigned long long);
const mp_float_gmp & my_value_inf(void) const;
const mp_float & my_value_max(void) const;
const mp_float & my_value_min(void) const;
const mp_float_gmp & my_value_nan(void) const;
mp_float_gmp & negate(void);
operator char() const;
operator double() const;
operator float() const;
operator int() const;
operator long double() const;
operator signed char() const;
operator signed long() const;
operator signed long long() const;
operator signed short() const;
operator unsigned char() const;
operator unsigned int() const;
operator unsigned long() const;
operator unsigned long long() const;
operator unsigned short() const;
operator wchar_t() const;
mp_float & operator*=(const mp_float &);
mp_float_gmp & operator*=(const mp_float_gmp &);
mp_float_gmp & operator++(void);
mp_float & operator+=(const mp_float &);
mp_float_gmp & operator+=(const mp_float_gmp &);
mp_float_gmp & operator--(void);
mp_float & operator-=(const mp_float &);
mp_float_gmp & operator-=(const mp_float_gmp &);
mp_float & operator/=(const mp_float &);
mp_float_gmp & operator/=(const mp_float_gmp &);
boost::int64_t order(void) const;
void precision(const boost::int32_t);
mp_float & sub_signed_long_long(const signed long);
mp_float_gmp & sub_unsigned_long_long(const unsigned long long);
void wr_string(std::string &, std::ostream &) const;

// private member functions
void from_unsigned_long(const unsigned long);
void from_unsigned_long_long(const unsigned long long);
boost::int64_t get_order_exact(void) const;
boost::int64_t get_order_fast(void) const;
void get_output_string(std::string &, boost::int64_t &, const std::size_t) const;
bool rd_string(const char *const);
```

```
  // private static functions
  static void init(void);
  static const boost::int64_t & max_exp2(void);
  static const boost::int64_t & min_exp2(void);

  // public static functions
  static bool char_is_nonzero_predicate(const char &);
  static mp_float my_acos(const mp_float &);
  static mp_float my_acosh(const mp_float &);
  static mp_float my_asin(const mp_float &);
  static mp_float my_asinh(const mp_float &);
  static mp_float my_atan(const mp_float &);
  static mp_float my_atanh(const mp_float &);
  static mp_float my_cbrt(const mp_float &);
  static mp_float my_cos(const mp_float &);
  static mp_float my_cosh(const mp_float &);
  static mp_float my_cyl_bessel_jn(const boost::int32_t, const mp_float &);
  static mp_float my_cyl_bessel_yn(const boost::int32_t, const mp_float &);
  static mp_float my_exp(const mp_float &);
  static mp_float my_fmod(const mp_float &, const mp_float &);
  static mp_float my_frexp(const mp_float &, int *);
  static mp_float my_gamma(const mp_float &);
  static mp_float my_ldexp(const mp_float &, int);
  static mp_float my_log(const mp_float &);
  static mp_float my_riemann_zeta(const mp_float &);
  static mp_float my_rootn(const mp_float &, const boost::uint32_t);
  static mp_float my_sin(const mp_float &);
  static mp_float my_sinh(const mp_float &);
  static mp_float my_tan(const mp_float &);
  static mp_float my_tanh(const mp_float &);

  // public data members
  static const boost::int32_t mp_float_digits;
  static const boost::int64_t mp_float_max_exp;
  static const boost::int64_t mp_float_max_exp10;
  static const boost::int64_t mp_float_min_exp;
  static const boost::int64_t mp_float_min_exp10;
  static const boost::int32_t mp_radix;
};
```

Description

`mp_float_gmp` public construct/copy/destruct

1. ```
   mp_float_gmp();
   ```

2. ```
   mp_float_gmp(const char n);
   ```

3. ```
   mp_float_gmp(const signed char n);
   ```

4. ```
   mp_float_gmp(const unsigned char n);
   ```

5. ```
   mp_float_gmp(const wchar_t n);
   ```
   ```

6. `mp_float_gmp(const signed short n);`
7. `mp_float_gmp(const unsigned short n);`
8. `mp_float_gmp(const int n);`
9. `mp_float_gmp(const unsigned int n);`
10. `mp_float_gmp(const signed long n);`
11. `mp_float_gmp(const unsigned long n);`
12. `mp_float_gmp(const signed long long n);`
13. `mp_float_gmp(const unsigned long long n);`
14. `mp_float_gmp(const float f);`
15. `mp_float_gmp(const double d);`
16. `mp_float_gmp(const long double ld);`
17. `mp_float_gmp(const char *const s);`
18. `mp_float_gmp(const std::string & str);`
19. `mp_float_gmp(const mp_float_gmp & f);`
20. `mp_float_gmp(const double mantissa, const boost::int64_t exponent);`
21. `explicit mp_float_gmp(const ::mpf_t & op);`

22. ```
mp_float_gmp& operator=(const mp_float_gmp & v);
```

23. ```
~mp_float_gmp();
```

`mp_float_gmp` public member functions

1. ```
mp_float & add_signed_long_long(const signed long long);
```

2. ```
mp_float_gmp & add_unsigned_long_long(const unsigned long long n);
```

3. ```
mp_float_gmp & calculate_inv(void);
```

4. ```
mp_float_gmp & calculate_sqrt(void);
```

5. ```
boost::int32_t cmp(const mp_float_gmp & v) const;
```

6. ```
boost::int32_t cmp(const mp_float &) const;
```

7. ```
mp_float & div_signed_long_long(const signed long long);
```

8. ```
mp_float_gmp & div_unsigned_long_long(const unsigned long long n);
```

9. ```
mp_float_gmp extract_decimal_part(void) const;
```

10. ```
double extract_double(void) const;
```

11. ```
mp_float_gmp extract_integer_part(void) const;
```

12. ```
long double extract_long_double(void) const;
```

13. ```
void extract_parts(double & mantissa, boost::int64_t & exponent) const;
```

14. ```
signed long long extract_signed_long_long(void) const;
```

```
15. unsigned long long extract_unsigned_long_long(void) const;

16. bool has_its_own_acos(void) const;

17. bool has_its_own_acosh(void) const;

18. bool has_its_own_asin(void) const;

19. bool has_its_own_asinh(void) const;

20. bool has_its_own_atan(void) const;

21. bool has_its_own_atanh(void) const;

22. bool has_its_own_cbrt(void) const;

23. bool has_its_own_cos(void) const;

24. bool has_its_own_cosh(void) const;

25. bool has_its_own_cyl_bessel_jn(void) const;

26. bool has_its_own_cyl_bessel_yn(void) const;

27. bool has_its_own_exp(void) const;

28. bool has_its_own_fmod(void) const;

29. bool has_its_own_frexp(void) const;

30. bool has_its_own_gamma(void) const;
```

```
31. bool has_its_own_ldexp(void) const;
32. bool has_its_own_log(void) const;
33. bool has_its_own_riemann_zeta(void) const;
34. bool has_its_own_rootn(void) const;
35. bool has_its_own_sin(void) const;
36. bool has_its_own_sinh(void) const;
37. bool has_its_own_tan(void) const;
38. bool has_its_own_tanh(void) const;
39. bool isfinite(void) const;
40. bool isinf(void) const;
41. bool isint(void) const;
42. bool isnan(void) const;
43. bool isneg(void) const;
44. bool isone(void) const;
45. bool ispos(void) const;
46. bool iszero(void) const;
```

```
47. mp_float & mul_signed_long_long(const signed long long);

48. mp_float_gmp & mul_unsigned_long_long(const unsigned long long n);

49. const mp_float_gmp & my_value_inf(void) const;

50. const mp_float & my_value_max(void) const;

51. const mp_float & my_value_min(void) const;

52. const mp_float_gmp & my_value_nan(void) const;

53. mp_float_gmp & negate(void);

54. operator char() const;

55. operator double() const;

56. operator float() const;

57. operator int() const;

58. operator long double() const;

59. operator signed char() const;

60. operator signed long() const;

61. operator signed long long() const;

62. operator signed short() const;
```

```
63. operator unsigned char() const;

64. operator unsigned int() const;

65. operator unsigned long() const;

66. operator unsigned long long() const;

67. operator unsigned short() const;

68. operator wchar_t() const;

69. mp_float & operator*=(const mp_float &);

70. mp_float_gmp & operator*=(const mp_float_gmp & v);

71. mp_float_gmp & operator++(void);

72. mp_float & operator+=(const mp_float &);

73. mp_float_gmp & operator+=(const mp_float_gmp & v);

74. mp_float_gmp & operator--(void);

75. mp_float & operator-=(const mp_float &);

76. mp_float_gmp & operator-=(const mp_float_gmp & v);

77. mp_float & operator/=(const mp_float &);

78. mp_float_gmp & operator/=(const mp_float_gmp & v);
```

79.
```
boost::int64_t order(void) const;
```

80.
```
void precision(const boost::int32_t prec_digits);
```

81.
```
mp_float & sub_signed_long_long(const signed long long);
```

82.
```
mp_float_gmp & sub_unsigned_long_long(const unsigned long long n);
```

83.
```
void wr_string(std::string & str, std::ostream & os) const;
```

`mp_float_gmp` private member functions

1.
```
void from_unsigned_long(const unsigned long u);
```

2.
```
void from_unsigned_long_long(const unsigned long long u);
```

3.
```
boost::int64_t get_order_exact(void) const;
```

4.
```
boost::int64_t get_order_fast(void) const;
```

5.
```
void get_output_string(std::string & str, boost::int64_t & my_exp,
                       const std::size_t number_of_digits) const;
```

6.
```
bool rd_string(const char *const s);
```

`mp_float_gmp` private static functions

1.
```
static void init(void);
```

2.
```
static const boost::int64_t & max_exp2(void);
```

3.
```
static const boost::int64_t & min_exp2(void);
```

`mp_float_gmp` public static functions

1.
```
static bool char_is_nonzero_predicate(const char & c);
```

```
2. static mp_float my_acos(const mp_float &);
3. static mp_float my_acosh(const mp_float &);
4. static mp_float my_asin(const mp_float &);
5. static mp_float my_asinh(const mp_float &);
6. static mp_float my_atan(const mp_float &);
7. static mp_float my_atanh(const mp_float &);
8. static mp_float my_cbrt(const mp_float &);
9. static mp_float my_cos(const mp_float &);
10. static mp_float my_cosh(const mp_float &);
11. static mp_float my_cyl_bessel_jn(const boost::int32_t, const mp_float &);
12. static mp_float my_cyl_bessel_yn(const boost::int32_t, const mp_float &);
13. static mp_float my_exp(const mp_float &);
14. static mp_float my_fmod(const mp_float &, const mp_float &);
15. static mp_float my_frexp(const mp_float &, int *);
16. static mp_float my_gamma(const mp_float &);
17. static mp_float my_ldexp(const mp_float &, int);
```

```
18. static mp_float my_log(const mp_float &);

19. static mp_float my_riemann_zeta(const mp_float &);

20. static mp_float my_rootn(const mp_float &, const boost::uint32_t);

21. static mp_float my_sin(const mp_float &);

22. static mp_float my_sinh(const mp_float &);

23. static mp_float my_tan(const mp_float &);

24. static mp_float my_tanh(const mp_float &);
```

Header <boost/multiprecision/mp_float_mpfr.hpp>

```
__gmp_const
mp_prec_t
mp_rnd_t
```

```
struct __mpfr_struct;namespace boost {
  namespace multiprecision {
    class mp_float_mpfr;
  }
}

typedef long int mp_exp_t;
typedef unsigned long int mp_limb_t;
typedef unsigned long mpfr_prec_t;
typedef __mpfr_struct * mpfr_ptr;
typedef int mpfr_sign_t;
typedef __gmp_const __mpfr_struct * mpfr_srcptr;
typedef __mpfr_struct mpfr_t;
```

Struct __mpfr_struct

__mpfr_struct

Synopsis

```
// In header: <boost/multiprecision/mp_float_mpfr.hpp>


struct __mpfr_struct {

  // public data members
  mp_limb_t * _mpfr_d;
  mp_exp_t _mpfr_exp;
  mpfr_prec_t _mpfr_prec;
  mpfr_sign_t _mpfr_sign;
};
```

Class mp_float_mpfr

boost::multiprecision::mp_float_mpfr

Synopsis

```
// In header: <boost/multiprecision/mp_float_mpfr.hpp>


class mp_float_mpfr : public boost::multiprecision::mp_float_base {
public:
  // construct/copy/destruct
  mp_float_mpfr();
  mp_float_mpfr(const char);
  mp_float_mpfr(const signed char);
  mp_float_mpfr(const unsigned char);
  mp_float_mpfr(const wchar_t);
  mp_float_mpfr(const signed short);
  mp_float_mpfr(const unsigned short);
  mp_float_mpfr(const int);
  mp_float_mpfr(const unsigned int);
  mp_float_mpfr(const signed long);
  mp_float_mpfr(const unsigned long);
  mp_float_mpfr(const signed long long);
  mp_float_mpfr(const unsigned long long);
  mp_float_mpfr(const float);
  mp_float_mpfr(const double);
  mp_float_mpfr(const long double);
  mp_float_mpfr(const char *const);
  mp_float_mpfr(const std::string &);
  mp_float_mpfr(const mp_float_mpfr &);
  mp_float_mpfr(const double, const boost::int64_t);
  mp_float_mpfr& operator=(const mp_float_mpfr &);
  ~mp_float_mpfr();

  // public member functions
  mp_float & add_signed_long_long(const signed long);
  mp_float_mpfr & add_unsigned_long_long(const unsigned long long);
  mp_float_mpfr & calculate_inv(void);
  mp_float_mpfr & calculate_sqrt(void);
  boost::int32_t cmp(const mp_float_mpfr &) const;
  boost::int32_t cmp(const mp_float &) const;
  mp_float & div_signed_long_long(const signed long);
  mp_float_mpfr & div_unsigned_long_long(const unsigned long long);
  mp_float_mpfr extract_decimal_part(void) const;
  double extract_double(void) const;
  mp_float_mpfr extract_integer_part(void) const;
  long double extract_long_double(void) const;
  void extract_parts(double &, boost::int64_t &) const;
  signed long long extract_signed_long_long(void) const;
  unsigned long long extract_unsigned_long_long(void) const;
  bool has_its_own_acos(void) const;
  bool has_its_own_acosh(void) const;
  bool has_its_own_asin(void) const;
  bool has_its_own_asinh(void) const;
  bool has_its_own_atan(void) const;
  bool has_its_own_atanh(void) const;
  bool has_its_own_cbrt(void) const;
  bool has_its_own_cos(void) const;
  bool has_its_own_cosh(void) const;
  bool has_its_own_cyl_bessel_jn(void) const;
  bool has_its_own_cyl_bessel_yn(void) const;
  bool has_its_own_exp(void) const;
  bool has_its_own_fmod(void) const;
  bool has_its_own_frexp(void) const;
  bool has_its_own_gamma(void) const;
```

```
bool has_its_own_ldexp(void) const;
bool has_its_own_log(void) const;
bool has_its_own_riemann_zeta(void) const;
bool has_its_own_rootn(void) const;
bool has_its_own_sin(void) const;
bool has_its_own_sinh(void) const;
bool has_its_own_tan(void) const;
bool has_its_own_tanh(void) const;
bool isfinite(void) const;
bool isinf(void) const;
bool isint(void) const;
bool isnan(void) const;
bool isneg(void) const;
bool isone(void) const;
bool ispos(void) const;
bool iszero(void) const;
mp_float & mul_signed_long_long(const signed long);
mp_float_mpfr & mul_unsigned_long_long(const unsigned long long);
const mp_float_mpfr & my_value_inf(void) const;
const mp_float & my_value_max(void) const;
const mp_float & my_value_min(void) const;
const mp_float_mpfr & my_value_nan(void) const;
mp_float_mpfr & negate(void);
operator char() const;
operator double() const;
operator float() const;
operator int() const;
operator long double() const;
operator signed char() const;
operator signed long() const;
operator signed long long() const;
operator signed short() const;
operator unsigned char() const;
operator unsigned int() const;
operator unsigned long() const;
operator unsigned long long() const;
operator unsigned short() const;
operator wchar_t() const;
mp_float & operator*=(const mp_float &);
mp_float_mpfr & operator*=(const mp_float_mpfr &);
mp_float_mpfr & operator++(void);
mp_float & operator+=(const mp_float &);
mp_float_mpfr & operator+=(const mp_float_mpfr &);
mp_float_mpfr & operator--(void);
mp_float & operator-=(const mp_float &);
mp_float_mpfr & operator-=(const mp_float_mpfr &);
mp_float & operator/=(const mp_float &);
mp_float_mpfr & operator/=(const mp_float_mpfr &);
boost::int64_t order(void) const;
void precision(const boost::int32_t);
mp_float & sub_signed_long_long(const signed long);
mp_float_mpfr & sub_unsigned_long_long(const unsigned long long);
void wr_string(std::string &, std::ostream &) const;

// public static functions
static bool char_is_nonzero_predicate(const char &);
static mp_float_mpfr my_acos(const mp_float_mpfr &);
static mp_float my_acos(const mp_float &);
static mp_float_mpfr my_acosh(const mp_float_mpfr &);
static mp_float my_acosh(const mp_float &);
static mp_float_mpfr my_asin(const mp_float_mpfr &);
static mp_float my_asin(const mp_float &);
static mp_float_mpfr my_asinh(const mp_float_mpfr &);
```

```
  static mp_float my_asinh(const mp_float &);
  static mp_float_mpfr my_atan(const mp_float_mpfr &);
  static mp_float my_atan(const mp_float &);
  static mp_float_mpfr my_atanh(const mp_float_mpfr &);
  static mp_float my_atanh(const mp_float &);
  static mp_float_mpfr my_cbrt(const mp_float_mpfr &);
  static mp_float my_cbrt(const mp_float &);
  static mp_float_mpfr my_cos(const mp_float_mpfr &);
  static mp_float my_cos(const mp_float &);
  static mp_float_mpfr my_cosh(const mp_float_mpfr &);
  static mp_float my_cosh(const mp_float &);
  static mp_float_mpfr
  my_cyl_bessel_jn(const boost::int32_t, const mp_float_mpfr &);
  static mp_float my_cyl_bessel_jn(const boost::int32_t, const mp_float &);
  static mp_float_mpfr
  my_cyl_bessel_yn(const boost::int32_t, const mp_float_mpfr &);
  static mp_float my_cyl_bessel_yn(const boost::int32_t, const mp_float &);
  static mp_float_mpfr my_exp(const mp_float_mpfr &);
  static mp_float my_exp(const mp_float &);
  static mp_float my_fmod(const mp_float &, const mp_float &);
  static mp_float my_frexp(const mp_float &, int *);
  static mp_float_mpfr my_gamma(const mp_float_mpfr &);
  static mp_float my_gamma(const mp_float &);
  static mp_float my_ldexp(const mp_float &, int);
  static mp_float_mpfr my_log(const mp_float_mpfr &);
  static mp_float my_log(const mp_float &);
  static mp_float_mpfr my_riemann_zeta(const mp_float_mpfr &);
  static mp_float my_riemann_zeta(const mp_float &);
  static mp_float_mpfr my_rootn(const mp_float_mpfr &, const boost::uint32_t);
  static mp_float my_rootn(const mp_float &, const boost::uint32_t);
  static mp_float_mpfr my_sin(const mp_float_mpfr &);
  static mp_float my_sin(const mp_float &);
  static mp_float_mpfr my_sinh(const mp_float_mpfr &);
  static mp_float my_sinh(const mp_float &);
  static mp_float_mpfr my_tan(const mp_float_mpfr &);
  static mp_float my_tan(const mp_float &);
  static mp_float_mpfr my_tanh(const mp_float_mpfr &);
  static mp_float my_tanh(const mp_float &);

  // private static functions
  static void init(void);

  // private member functions
  void from_unsigned_long(const unsigned long);
  void from_unsigned_long_long(const unsigned long long);
  boost::int64_t get_order_exact(void) const;
  boost::int64_t get_order_fast(void) const;
  void get_output_string(std::string &, boost::int64_t &, const std::size_t) const;
  bool rd_string(const char *const);

  // public data members
  static const boost::int32_t mp_float_digits;
  static const boost::int64_t mp_float_max_exp;
  static const boost::int64_t mp_float_max_exp10;
  static const boost::int64_t mp_float_min_exp;
  static const boost::int64_t mp_float_min_exp10;
  static const boost::int32_t mp_radix;
};
```

Description

`mp_float_mpfr` public construct/copy/destruct

1. ```
   mp_float_mpfr();
   ```

2. ```
   mp_float_mpfr(const char n);
   ```

3. ```
   mp_float_mpfr(const signed char n);
   ```

4. ```
   mp_float_mpfr(const unsigned char n);
   ```

5. ```
   mp_float_mpfr(const wchar_t n);
   ```

6. ```
   mp_float_mpfr(const signed short n);
   ```

7. ```
   mp_float_mpfr(const unsigned short n);
   ```

8. ```
   mp_float_mpfr(const int n);
   ```

9. ```
   mp_float_mpfr(const unsigned int n);
   ```

10. ```
    mp_float_mpfr(const signed long n);
    ```

11. ```
    mp_float_mpfr(const unsigned long n);
    ```

12. ```
    mp_float_mpfr(const signed long long n);
    ```

13. ```
    mp_float_mpfr(const unsigned long long n);
    ```

14. ```
    mp_float_mpfr(const float f);
    ```

15. ```
    mp_float_mpfr(const double d);
    ```
    ```

16. `mp_float_mpfr(const long double ld);`

17. `mp_float_mpfr(const char *const s);`

18. `mp_float_mpfr(const std::string & str);`

19. `mp_float_mpfr(const mp_float_mpfr & f);`

20. `mp_float_mpfr(const double mantissa, const boost::int64_t exponent);`

21. `mp_float_mpfr& operator=(const mp_float_mpfr & v);`

22. `~mp_float_mpfr();`

`mp_float_mpfr` public member functions

1. `mp_float & add_signed_long_long(const signed long long);`

2. `mp_float_mpfr & add_unsigned_long_long(const unsigned long long n);`

3. `mp_float_mpfr & calculate_inv(void);`

4. `mp_float_mpfr & calculate_sqrt(void);`

5. `boost::int32_t cmp(const mp_float_mpfr & v) const;`

6. `boost::int32_t cmp(const mp_float &) const;`

7. `mp_float & div_signed_long_long(const signed long long);`

8. `mp_float_mpfr & div_unsigned_long_long(const unsigned long long n);`

9. `mp_float_mpfr extract_decimal_part(void) const;`

```
10. double extract_double(void) const;
11. mp_float_mpfr extract_integer_part(void) const;
12. long double extract_long_double(void) const;
13. void extract_parts(double & mantissa, boost::int64_t & exponent) const;
14. signed long long extract_signed_long_long(void) const;
15. unsigned long long extract_unsigned_long_long(void) const;
16. bool has_its_own_acos(void) const;
17. bool has_its_own_acosh(void) const;
18. bool has_its_own_asin(void) const;
19. bool has_its_own_asinh(void) const;
20. bool has_its_own_atan(void) const;
21. bool has_its_own_atanh(void) const;
22. bool has_its_own_cbrt(void) const;
23. bool has_its_own_cos(void) const;
24. bool has_its_own_cosh(void) const;
25. bool has_its_own_cyl_bessel_jn(void) const;
```

```
bool has_its_own_cyl_bessel_yn(void) const;
```

```
bool has_its_own_exp(void) const;
```

```
bool has_its_own_fmod(void) const;
```

```
bool has_its_own_frexp(void) const;
```

```
bool has_its_own_gamma(void) const;
```

```
bool has_its_own_ldexp(void) const;
```

```
bool has_its_own_log(void) const;
```

```
bool has_its_own_riemann_zeta(void) const;
```

```
bool has_its_own_rootn(void) const;
```

```
bool has_its_own_sin(void) const;
```

```
bool has_its_own_sinh(void) const;
```

```
bool has_its_own_tan(void) const;
```

```
bool has_its_own_tanh(void) const;
```

```
bool isfinite(void) const;
```

```
bool isinf(void) const;
```

```
bool isint(void) const;
```

```
42. bool isnan(void) const;

43. bool isneg(void) const;

44. bool isone(void) const;

45. bool ispos(void) const;

46. bool iszero(void) const;

47. mp_float & mul_signed_long_long(const signed long long);

48. mp_float_mpfr & mul_unsigned_long_long(const unsigned long long n);

49. const mp_float_mpfr & my_value_inf(void) const;

50. const mp_float & my_value_max(void) const;

51. const mp_float & my_value_min(void) const;

52. const mp_float_mpfr & my_value_nan(void) const;

53. mp_float_mpfr & negate(void);

54. operator char() const;

55. operator double() const;

56. operator float() const;

57. operator int() const;
```

```
operator long double() const;
```

```
operator signed char() const;
```

```
operator signed long() const;
```

```
operator signed long long() const;
```

```
operator signed short() const;
```

```
operator unsigned char() const;
```

```
operator unsigned int() const;
```

```
operator unsigned long() const;
```

```
operator unsigned long long() const;
```

```
operator unsigned short() const;
```

```
operator wchar_t() const;
```

```
mp_float & operator*=(const mp_float &);
```

```
mp_float_mpfr & operator*=(const mp_float_mpfr & v);
```

```
mp_float_mpfr & operator++(void);
```

```
mp_float & operator+=(const mp_float &);
```

```
mp_float_mpfr & operator+=(const mp_float_mpfr & v);
```

74. `mp_float_mpfr & operator--(void);`

75. `mp_float & operator-=(const mp_float &);`

76. `mp_float_mpfr & operator-=(const mp_float_mpfr & v);`

77. `mp_float & operator/=(const mp_float &);`

78. `mp_float_mpfr & operator/=(const mp_float_mpfr & v);`

79. `boost::int64_t order(void) const;`

80. `void precision(const boost::int32_t);`

81. `mp_float & sub_signed_long_long(const signed long long);`

82. `mp_float_mpfr & sub_unsigned_long_long(const unsigned long long n);`

83. `void wr_string(std::string & str, std::ostream & os) const;`

`mp_float_mpfr` public static functions

1. `static bool char_is_nonzero_predicate(const char & c);`

2. `static mp_float_mpfr my_acos(const mp_float_mpfr & x);`

3. `static mp_float my_acos(const mp_float &);`

4. `static mp_float_mpfr my_acosh(const mp_float_mpfr & x);`

5. `static mp_float my_acosh(const mp_float &);`

6. `static mp_float_mpfr my_asin(const mp_float_mpfr & x);`

```
7.  static mp_float my_asin(const mp_float &);
8.  static mp_float_mpfr my_asinh(const mp_float_mpfr & x);
9.  static mp_float my_asinh(const mp_float &);
10. static mp_float_mpfr my_atan(const mp_float_mpfr & x);
11. static mp_float my_atan(const mp_float &);
12. static mp_float_mpfr my_atanh(const mp_float_mpfr & x);
13. static mp_float my_atanh(const mp_float &);
14. static mp_float_mpfr my_cbrt(const mp_float_mpfr & x);
15. static mp_float my_cbrt(const mp_float &);
16. static mp_float_mpfr my_cos(const mp_float_mpfr & x);
17. static mp_float my_cos(const mp_float &);
18. static mp_float_mpfr my_cosh(const mp_float_mpfr & x);
19. static mp_float my_cosh(const mp_float &);
20. static mp_float_mpfr
    my_cyl_bessel_jn(const boost::int32_t n, const mp_float_mpfr & x);
21. static mp_float my_cyl_bessel_jn(const boost::int32_t, const mp_float &);
22. static mp_float_mpfr
    my_cyl_bessel_yn(const boost::int32_t n, const mp_float_mpfr & x);
```

```
23. static mp_float my_cyl_bessel_yn(const boost::int32_t, const mp_float &);
24. static mp_float_mpfr my_exp(const mp_float_mpfr & x);
25. static mp_float my_exp(const mp_float &);
26. static mp_float my_fmod(const mp_float &, const mp_float &);
27. static mp_float my_frexp(const mp_float &, int *);
28. static mp_float_mpfr my_gamma(const mp_float_mpfr & x);
29. static mp_float my_gamma(const mp_float &);
30. static mp_float my_ldexp(const mp_float &, int);
31. static mp_float_mpfr my_log(const mp_float_mpfr & x);
32. static mp_float my_log(const mp_float &);
33. static mp_float_mpfr my_riemann_zeta(const mp_float_mpfr & x);
34. static mp_float my_riemann_zeta(const mp_float &);
35. static mp_float_mpfr
    my_rootn(const mp_float_mpfr & x, const boost::uint32_t p);
36. static mp_float my_rootn(const mp_float &, const boost::uint32_t);
37. static mp_float_mpfr my_sin(const mp_float_mpfr & x);
38. static mp_float my_sin(const mp_float &);
```

39.
```
static mp_float_mpfr my_sinh(const mp_float_mpfr & x);
```

40.
```
static mp_float my_sinh(const mp_float &);
```

41.
```
static mp_float_mpfr my_tan(const mp_float_mpfr & x);
```

42.
```
static mp_float my_tan(const mp_float &);
```

43.
```
static mp_float_mpfr my_tanh(const mp_float_mpfr & x);
```

44.
```
static mp_float my_tanh(const mp_float &);
```

`mp_float_mpfr` private static functions

1.
```
static void init(void);
```

`mp_float_mpfr` private member functions

1.
```
void from_unsigned_long(const unsigned long u);
```

2.
```
void from_unsigned_long_long(const unsigned long long u);
```

3.
```
boost::int64_t get_order_exact(void) const;
```

4.
```
boost::int64_t get_order_fast(void) const;
```

5.
```
void get_output_string(std::string & str, boost::int64_t & my_exp,
                       const std::size_t number_of_digits) const;
```

6.
```
bool rd_string(const char *const s);
```

Macro __gmp_const

__gmp_const

Synopsis

```
// In header: <boost/multiprecision/mp_float_mpfr.hpp>

__gmp_const
```

Macro mp_prec_t

mp_prec_t

Synopsis

```
// In header: <boost/multiprecision/mp_float_mpfr.hpp>

mp_prec_t
```

Macro mp_rnd_t

mp_rnd_t

Synopsis

```
// In header: <boost/multiprecision/mp_float_mpfr.hpp>

mp_rnd_t
```

Class Index

Symbols

B

M

Typedef Index

Symbols

M

Function Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 90, 91
my_asinh
Header < boost/multiprecision/mp_float_base.hpp >, 37, 44
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91
my_atan
Header < boost/multiprecision/mp_float_base.hpp >, 37, 44
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91
my_atanh
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91
my_cbrt
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91
my_cos
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91
my_cosh
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91
my_cyl_bessel_jn
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91
my_cyl_bessel_yn
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91, 92
my_exp
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 92
my_fmod
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 92
my_frexp
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 92
my_gamma

N

O

P

Q

R

S

T

U

V

Z

Macro Index

Symbols

B

M

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 88
my_acos
Header < boost/multiprecision/mp_float_base.hpp >, 37, 44
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 90
my_acosh
Header < boost/multiprecision/mp_float_base.hpp >, 37, 44
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 90
my_asin
Header < boost/multiprecision/mp_float_base.hpp >, 37, 44
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 90, 91
my_asinh
Header < boost/multiprecision/mp_float_base.hpp >, 37, 44
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91
my_atan
Header < boost/multiprecision/mp_float_base.hpp >, 37, 44
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91
my_atanh
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91
my_cbrt
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91
my_cos
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91
my_cosh
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91
my_cyl_bessel_jn
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91
my_cyl_bessel_yn
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 59
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 91, 92
my_exp
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45

Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 92
my_fmod
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 92
my_frexp
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 92
my_gamma
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 92
my_ldexp
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 78
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 92
my_log
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 79
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 92
my_riemann_zeta
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 79
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 92
my_rootn
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 79
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 92
my_sin
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 79
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 92
my_sinh
Header < boost/multiprecision/mp_float_base.hpp >, 37, 45
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 79
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 93
my_tan
Header < boost/multiprecision/mp_float_base.hpp >, 37, 46
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 79
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 93
my_tanh
Header < boost/multiprecision/mp_float_base.hpp >, 37, 46
Header < boost/multiprecision/mp_float_efx.hpp >, 49, 60
Header < boost/multiprecision/mp_float_gmp.hpp >, 68, 79
Header < boost/multiprecision/mp_float_mpfr.hpp >, 81, 93

N

O

P

Q

R

S

T

U

V

W

Z