Boost.LargeInt

Michael Tryhorn

Copyright © 2012 Michael J. Tryhorn

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

Overview

Boost.LargeInt is a platform-independent C++ library for the definition and transparent manipulation of integers of arbitrary size. If you are looking for 128bit integer support only, feel free to skip to the quick start guide.

Component	Header	Purpose
Forward Declarations	`<boost/large_int/fwd.hpp>`	Forward declarations of classes, class templates, functions and operators - for use when only an identifier is needed.
Full declarations	`<boost/large_int.hpp>`	All large_int declarations and their implementation **except for** the implementations for the streaming operators: << and >>.
Large Integer	`<boost/large_int/base.hpp>`	The `boost::large_int::large_int` class declaration itself, including a complete set of constructors (with implementation) and operator forward declarations.
Large Integer Cast	`<boost/large_int/cast.hpp>`	The custom cast operator `boost::large_int::large_int_cast`, for casting to, from and between `large_int` values.
Large Integer Divide	`<boost/large_int/div.hpp>`	The standard arithmetic function `boost::large_int::div` and associated result type `boost::large_int::large_int_div_t`, to be used where both division and modulus of a large_int value are required.
Large Integer Limits	`<boost/large_int/limits.hpp>`	A partial-specialisation of class template `std::numeric_limits` for all `boost::large_int::large_int` types.
Large Integer Traits	`<boost/large_int/traits.hpp>`	Class template `boost::large_int::large_int_traits`, including static members for the detection, inspection and comparison of `large_int` values.
Large Integer Utils	`<boost/large_int/utils.hpp>`	Utility function templates, including `boost::large_int::abs` and `boost::large_int::make_large_int`.
Large Integer Defaults	`<boost/large_int/def.hpp>`	Standard `boost::large_int::large_int` based integral types.
Large Integer I/O	`<boost/large_int/io.hpp>`	Implementations for streaming operators << and >> for all `boost::large_int::large_int` based integral types.
Large Integer Library Version	`<boost/large_int/version.hpp>`	The version of the Boost.LargeInt library, as a defined value.

Caveat Emptor

This library makes extensive use of template-metaprogramming, and as such places a heavy demand upon the C++ compiler. Although every effort has been made to retain code-portability, not all compilers will be able to support code which itself relies upon Boost.LargeInt. However, this library has been tested with the following compilers:

Compiler	Versions	Library Status
GCC	4.6.1, 4.6.2	Fully functional
Borland C++	5.6.4	Fully functional
Microsoft Visual C++	2010	Fully functional
Microsoft Visual C++	6	**Non-functional**

Quick Start Guide for 128bit Mathematicians

The Boost.LargeInt library was initially designed solely to provide a simple means to perform 128bit integral mathematics where native compiler support is unavailable. If this is all you require, a subset of the Boost.LargeInt functionality is all that must be learnt.

128bit integral mathematics is commonly required where the result must be calculated of the multiplication of two 64bit signed or unsigned lesser values. Either we restrict the range of the input values - for example by throwing an exception if their combination would overflow 64bits - or we assume that overflow will never occur. For example:

```
#include <stdexcept> // For std::range_error
#include <boost/cstdint.hpp> // For uint64_t

uint64_t do_square(uint64_t x)
{
    // Ensure that the calculation will not overflow
    const uint64_t sqrt_uint64_max(4294967295); // sqrt(2^64)-1
    if( x > sqrt_uint64_max )
    {
        // Throw an exception rather than produce an incorrect result
        throw std::range_error("Overflow in do_square");
    }
    return( x * x );
}
```

With Boost.LargeInt we can be sure that it will never occur. By promoting the input to 128bit before calculating its square, we could calculate the square of any integer in the range: [0...2^{64}-1]. Enter `boost::large_int::luint128_t`:

```
#include <boost/cstdint.hpp> // For uint64_t
#include <boost/large_int.hpp> // For boost::large_int::luint128_t

boost::large_int::luint128_t do_square(uint64_t x)
{
    // Promote the input to 128bit for multiplication
    return( boost::large_int::luint128_t(x)
          * boost::large_int::luint128_t(x) );
}
```

What about when we need to convert back from a 128bit integer to 64bit (or less)? The standard `static_cast` operator cannot be used. Instead, Boost.LargeInt provides its own, `static_cast`-like operator, `boost::large_int::large_int_cast`:

```
#include <boost/cstdint.hpp> // For uint64_t
#include <boost/large_int.hpp> // For boost::large_int::luint128_t
```

```
                                //     boost::large_int::large_int_cast
uint64_t do_sqrt(boost::large_int::luint128_t x)
{
    // [-- Perform the square root calculation ---]

    // -'x' is now in the range [0...(2^64)-1],
    // and may be safely cast to a 64bit result
    return boost::large_int::large_int_cast<uint64_t>(x);
}
```

Note

`boost::large_int::large_int_cast` supports the conversion to, from, and between `large_int` values. For fully-generic programming, it even supports conversion between two non-`large_int` values. Thus, in the context of Boost.LargeInt, it may be thought of as a replacement for `static_cast`.

Now, what if we want to write a simple application to take in a 64bit value, find its square, then echo that square back to the user? Simple:

```
#include <cstdlib> // For EXIT_SUCCESS, EXIT_FAILURE
#include <iostream> // For std::cin, std::cout, std::cerr,
                    //     std::flush, std::endl
#include <boost/cstdint.hpp> // For int64_t
#include <boost/large_int.hpp> // For boost::large_int::lint128_t
#include <boost/large_int/io.hpp> // For streaming operators << and >>

boost::large_int::luint128_t do_square(uint64_t x)
{
    // Promote the input to 128bit for multiplication
    return( boost::large_int::luint128_t(x)
          * boost::large_int::luint128_t(x) );
}

int main()
{
    int64_t val;
    std::cout << "Enter an integer: -" << std::flush;
    if( std::cin >> val )
    {
        // Square and echo the integer
        std::cout << "The square of -" << val
                  << " is -" << do_square(val)
                  << '.' << std::endl;
        return EXIT_SUCCESS;
    }
    else
    {
        // Bad input
        std::cerr << "Not an integer!" << std::endl;
        return EXIT_FAILURE;
    }
}
```

Warning

Where streaming input or output of a `boost::large_int::large_int` value is required, the I/O specific header must be explicitly included: `#include <boost/large_int/io.hpp>`.

This covers the basics. If you require more, please read on.

The Large Integer Object

Motivation

When defining functions with integers in standard, school-type mathematics, there are neither restrictions upon the input values that may be used, nor upon the result that may be calculated, except those limits which we impose ourselves (and, of course, the oft-lurking *division-by-zero*). There are also few restrictions upon the order of calculation, especially where the mathematical operators are commutative[1] or associative[2]. Not so in C, or C++.

When designing integral calculations in either C or C++, the programmer must be conscious of at least these three things:

1. the possible input value range,
2. the possible output value range,
3. the maximum or minimum value which could be stored during the calculation itself.

From these, a decision must be made as to the order of calculation and the size of the integers' storage. Should the input or output values be declared as: 'int', 'short', 'long', 'unsigned long', 'intmax_t', 'std::size_t', a template, or otherwise? Making an inappropriate decision here could be disastrous. For example, consider the following simple function for converting disk usage to percentage values.

In standard mathematics we might such a function thus:

$f(u, t) = 100(u / t)$; Where **u** is the bytes-used, **t** the bytes-total, and ***f*(u, t)** is the usage level as a percentage

And in C/C++, we might convert it to the following:

```
int disk_used_to_percent(int bytes_used, int bytes_total)
{
    return (bytes_used * 100) / bytes_total;
}
```

> **Note**
>
> The order of division and multiplication in the equivalent function is reversed as, in C/C++, neither the multiplication nor division operators are strictly associative. If these operations were not re-ordered, then the function would return `0` for all values of `bytes_used` in the range: (-`bytes_total`...+`bytes_total`).

For most input values, this function will return exactly the value expected, and if the input never exceeds the limits of the 'int' type, this may indeed suffice. However, if the compiler represents 'int' as 32bit (including one bit for the integer's sign), and if the user happens to have a 2GiB system disk (2^{31} bytes), what would occur when the usage is 50%?

```
return (1073741824 * 100 /* Overflow! */) / 2147483648;
```

The integer overflows! We might receive a result of 0%, but definitely not the expected 50%. There is a common trick to work-around this precise problem. If we re-write the function thus:

```
#include <boost/cstdint.hpp> // For intmax_t

int disk_used_to_percent(int bytes_used, int bytes_total)
{
    return (static_cast<intmax_t>(bytes_used) * 100) / bytes_total;
}
```

[1] Commutative: For all values of 'a' and 'b', rearranging the terms around a mathematical operator produces the same result; a × b = b × a.
[2] Associative: For all values of 'a', 'b' and 'c', changing the order of evaluation of a mathematical operation produces the same result; (a + b) + c = a + (b + c).

Provided that the compiler has support for integers larger than the standard 'int', we will both resolve the overflow problem and receive the expected result of 50%. But, what if the compiler *does not* support 64bit or larger integers? Or, what if we have the more modern scenario of having to write the function thus:

```
#include <boost/cstdint.hpp> // For uint64_t

int disk_used_to_percent(uint64_t bytes_used, uint64_t bytes_total)
{
    // Oops!  Cannot cast -'bytes_used' as
    // we have nothing larger than 64bit!
    return (bytes_used * 100) / bytes_total;
}
```

Again, for most input this will be fine. But again, what if we are a data-centre manager and this is a large, logical disk? Or, what if we are a particularly prudent programmer and want to ensure that the function will always produce the expected result *no matter what* input values are used? A way of manipulating integers larger than 64bits in size is needed, but what?

1. We might find something compiler or system specific - but even if something were available, using it would reduce the function's portability.
2. We might re-write the function to use floating-point arithmetic (`float`, or `double`), for its extended range - but this would add the inaccuracy inherent in floating-point values. (This would be more profound in examples more complex than this, but is still a valid concern.)
3. We might even consider encoding the values as Binary Coded Decimal [3].
4. What we really need is a means of manipulating integers larger than those the compiler natively supports - what we need is Boost.LargeInt!

Defining a New LargeInt Based Type

Boost.LargeInt provides the means to declare and manipulate integer values which are larger than the those the underlying compiler natively supports. It achieves this by composing two smaller integers into a larger whole.

To use Boost.LargeInt, a type of `boost::large_int::large_int<T, U>` must be defined, where:

`T` The low-part (LSB) of the large_int value. This **must** be both *unsigned*, and an *integer* or *integer-like* type.

`U` The high-part (MSB) of the large_int value. This **may** be either *signed* or *unsigned*, but **must** be an *integer* or *integer-like* type.

So, to declare a 128bit integer for a compiler which supports 64bit natively, we might declare a our new integer thus:

```
// Declare a 128bit signed integer composed of two 64s
typedef boost::large_int::large_int<uint64_t, int64_t> lint128_t;
```

As Boost.LargeInt defines all standard operators for this type, we may now use it in all calculations where 128 binary bits of integral space are required. If we were performing multiplications with 128bit values, we might even require more than 128bits to store the intermediate results. If so, the solution is just as simple:

```
// Declare a 256bit signed integer composed
// of two 128s, themselves composed of two 64s apiece
typedef boost::large_int::large_int<uint64_t, int64_t> lint128_t;
typedef boost::large_int::large_int<uint64_t, uint64_t> luint128_t;
typedef boost::large_int::large_int<luint128_t, lint128_t> lint256_t;
```

As `boost::large_int::large_int` types are themselves *integer-like*, they may be used as the low-part (**`T`**), high-part (**`U`**) or both parts of yet another `boost::large_int::large_int` based type, thus removing any limits upon the possible size of these integers except for what the compiler or system must necessarily impose themselves. (For example: there may be a compiler-imposed limit on maximum template recursion.) There is also no requirement that the low and high parts match in size. If we prefer, we could go half-way:

```
// Declare a 192bit integer composed of one 128s and one 64
typedef boost::large_int::large_int<uint64_t, int64_t> lint128_t;
typedef boost::large_int::large_int<luint128_t, int64_t> lint192_t;
```

Note

As the size of any `boost::large_int::large_int` type increases, so must the time-complexity of its multiplication, division and modulus operations (as they would have additional binary-bits to manipulate). By defining a more economically-sized type, we may improve calculation performance.

Specifically, for a type to be sufficiently *integer-like* and therefore usable as part of a new `large_int`, the type must support the following:

Operation	**Description**
`operator!`	Logical NOT
`operator++` (prefix)	Increment by 1
`operator--` (prefix)	Decrement by 1
`operator~`	Bitwise compliment
`operator|`	Bitwise OR
`operator&`	Bitwise AND
`operator^`	Bitwise XOR
`operator<<`	Bitwise left-shift
`operator>>`	Bitwise right-shift
`operator<<=`	Bitwise left-shift-and-store
`operator>>=`	Bitwise right-shift-and-store
`operator+=`	Add-and-store
`operator|=`	Bitwise OR-and-store
`operator<`	Logical less-than
`operator==`	Logical equals
`operator=`	Standard assignment
Default-construction	Construction from zero arguments
Copy-construction	Construction from another instance
std::numeric_limits	A specialization of std::numeric_limits must be available

> **Note**
>
> All built-in integers and those provided by cstdint or an equivalent support the above operations and more, and are therefore usable with Boost.LargeInt. Where any attempt is made to use an inappropriate type, the respective application code will likely fail to compile.

LargeInt Construction

Once we have a concrete `boost::large_int::large_int` based type, how may we create one or more instances and assign values to them? By far the the simplest way is by construction, and `boost::large_int::large_int` supports the following constructors:

```
explicit large_int(const T& lo = T());
large_int(const U& hi, const T& lo);
template<class Value> large_int(const Value& val);
```

The first acts as both the default, zero-argument constructor and as a single-argument constructor taking a value for the new `large_int`'s low-part. This constructor will always set the `large_int`'s high-part to zero, and yield a non-negative result.

```
// Declare and construct a pair of 128bit integers
typedef boost::large_int::large_int<uint64_t, int64_t> lint128_t;
lint128_t default_constructed;
lint128_t initialised_to_one(static_cast<uint64_t>(1));
```

The second allows the creation of a `large_int` by specifying the initial content for both the high and low parts.

```
// Declare and construct a 128bit integer, with fully initialised content
typedef boost::large_int::large_int<uint64_t, int64_t> lint128_t;
lint128_t fully_initialised(static_cast<int64_t>(0x0123456789abcdef),
                            static_cast<uint64_t>(0xfedcba9876543210));
```

The third and final constructor template allows for the creation of a `large_int` from any *integer-like* source. It is implicit, to allow the construction of `large_int` values from literal function or operator arguments.

```
// Declare and construct a 128bit integer from a literal source
typedef boost::large_int::large_int<uint64_t, int64_t> lint128_t;
lint128_t cast_constructed(-17);
```

All three constructors have constant, **O(1)** time-complexity.

> **Note**
>
> Although not explicitly specified, all `boost::large_int::large_int` types also support the default copy-constructor and assignment operator.

`boost::large_int::large_int` values may also be created via the custom casting operator `boost::large_int::large_int_cast`, via the utility function `boost::large_int::make_large_int`, or even via `boost::lexical_cast`. These are covered in later sections.

> **Note**
>
> For `boost::lexical_cast` support, the streaming operators must be made available via `#include <boost/large_int/io.hpp>`.

Large Integer Cast

For most common situations, any `boost::large_int::large_int` based type may be used as if it were a native equivalent. However, where a large_int value must be converted to another native or non-native type, the standard cast operators cannot be used. This is due to the fact that the standard casting operators cannot be overloaded and is a property of the C++ language itself.

However, we can easily work around this by defining our own cast operator: `boost::large_int::large_int_cast`. This custom cast is equivalent to (and may be used in place of) C++'s built-in `static_cast`.

```
namespace boost {
namespace large_int {

template<class C, class T> C large_int_cast(const T& val);

} // namespace large_int
} // namespace boost
```

> **Tip**
>
> Although we cannot overload the `static_cast` operator directly, it is possible to add casting operations to classes in the form of 'operator <TYPENAME>();'. However, this also implies that values of the source type may be *implicitly-cast* to the destination, and is therefore not an exact match to the explicit nature of `static_cast`.

Wherever a value must be converted either to or from a `boost::large_int::large_int` based type (or templatized type which *may or may not be* a `boost::large_int::large_int`), this conversion may be achieved via an instruction of the form:

```
// Cast the integer -'value' to type -'C'
C result = boost::large_int::large_int_cast<C>(value);
```

Where '**C**' is the desired destination type.

With `boost::large_int::large_int_cast` in hand, we may now re-write our original disk usage example for 64bit support, thus:

```
#include <boost/cstdint.hpp> // For uint64_t, int8_t
#include <boost/large_int.hpp> // For large_int, large_int_cast

int disk_used_to_percent(uint64_t bytes_used, uint64_t bytes_total)
{
    // Perform the calculation using 72bit values, which is large
    // enough to hold ((std::numeric_limits<uint64_t>::max)() * 100).
    typedef boost::large_int::large_int<uint64_t, int8_t> lint72_t;
    return boost::large_int::large_int_cast<int>(
        (lint72_t(bytes_used) * 100) / bytes_total);
}
```

Unlike the built-in cast operators we may, if we prefer, be completely explicit and specify the source type as a second template argument, after '**C**':

```
#include <boost/cstdint.hpp> // For uint64_t, int8_t
#include <boost/large_int.hpp> // For large_int, large_int_cast

int disk_used_to_percent(uint64_t bytes_used, uint64_t bytes_total)
{
    // Perform the usage calculating, specifying all
    // source and destination integer types explicitly
```

```
    typedef boost::large_int::large_int<uint64_t, int8_t> lint72_t;
    using boost::large_int::large_int_cast;
    return large_int_cast<int, lint72_t>(
          (large_int_cast<lint72_t, int>(bytes_used) * 100)
        / large_int_cast<lint72_t, int>(bytes_total));
}
```

Large Integer Divide

In integer arithmetic, where both a division and a modulus of a pair of terms are required we may combine them into a single call using the standard C/C++ functions: `std::div` or `std::ldiv`, and their respective result types: `std::div_t` and `std::ldiv_t`.

Boost.LargeInt also supports this behaviour, with the function template: `boost::large_int::div`, and a class template to hold its result: `boost::large_int::large_int_div_t`.

```
namespace boost {
namespace large_int {

template<class T, class U> struct large_int_div_t<large_int<T, U> >
{
    // Values ---
    large_int<T, U> quot; // The quotient
    large_int<T, U> rem; // The remainder

    // Constructor ---
    explicit large_int_div_t(
        const large_int<T, U>& quot_in = large_int<T, U>(),
        const large_int<T, U>& rem_in = large_int<T, U>());
};

template<class T, class U> large_int_div_t<large_int<T, U> > div(
    const large_int<T,U>& numerator, const large_int<T,U>& denominator);

} // namespace large_int
} // namespace boost
```

Tip

The `boost::large_int::large_int` operators `operator/`, `operator/=`, `operator%` and `operator%=` also use `boost::large_int::div` to perform their respective division operations. So, where both division and modulus are required, `boost::large_int::div` is highly recommended as its usage would reduce the cost in time by approximately half.

Usage Example

```
#include <iostream> // For std::cout, std::endl
#include <boost/cstdint.hpp> // For uint64_t, int8_t
#include <boost/large_int.hpp> // For boost::large_int::large_int,
                               //     boost::large_int::large_int_div_t,
                               //     boost::large_int::div
#include <boost/large_int/io.hpp> // For streaming operator<<

void example()
{
    // Calculate a quotient and modulus together
    typedef boost::large_int::large_int<uint64_t, int8_t> lint72_t;
    typedef boost::large_int::large_int_div_t<lint72_t> ldiv72_t;
```

```
    lint72_t numerator(7);
    lint72_t denominator(24);
    ldiv72_t result(boost::large_int::div(numerator, denominator));

    // Print the result
    std::cout << numerator << '/' << denominator << "==" << result.quot
              << std::endl;
    std::cout << numerator << '%' << denominator << "==" << result.rem
              << std::endl;
}
```

Large Integer Limits

To support generic programming, Boost.LargeInt also provides a specialization of the standard class '`std::numeric_limits`', for all `boost::large_int::large_int` types. This specialization may be used in exactly the same manner as for the built-in integral types.

```
namespace std
{

class numeric_limits< ::boost::large_int::large_int< /*...*/ > >
{
public:
    static const bool is_specialized = true;
    static const int digits = /*...*/;
    static const int digits10 = /*...*/;
    static const bool is_signed = /*...*/;
    static const bool is_integer = true;
    static const bool is_exact = true;
    static const int radix = 2;

    static ::boost::large_int::large_int< /*...*/ > min() throw();
    static ::boost::large_int::large_int< /*...*/ > max() throw();

    // -...
};

} // namespace std
```

Note

Although the definition for `std::numeric_limits` has changed between the C++98 and C++11 standards, Boost.LargeInt currently uses the C++98 version of the class definition to improve portability.

Tip

Specializations for classes in namespace `std` are explicitly allowed by the C++98 standard, section 17.4.3.1 ("Reserved names").

Usage Example

```
#include <iostream> // For std::cout, std::endl
#include <boost/cstdint.hpp> // For uint64_t, int8_t
#include <boost/large_int.hpp> // For boost::large_int::large_int,
                               //     std::numeric_limits<large_int>
void example()
```

```
{
    // Print the size, in bits, of a 72bit large_int
    typedef boost::large_int::large_int<uint64_t, int8_t> lint72_t;
    std::cout << "lint72_t has -" << std::numeric_limits<lint72_t>::digits
              << " binary digits" << std::endl;
}
```

Large Integer Traits

To support generic programming, Boost.LargeInt also defines the class template `boost::large_int::large_int_traits`. This is fully defined for all `large_int` types.

```
namespace boost {
namespace large_int {

template<class T> struct large_int_traits
{
public:
    // Types ---
    typedef /*...*/ low_part_type;
    typedef /*...*/ high_part_type;

    // Constants ---
    static const bool is_large_int = true;
    static const int low_bits = /*...*/;
    static const int high_bits = /*...*/;
    static const int size_bits = /*...*/;

    // Utilities ---
    template<class T2, class U2>
    static bool lt(const large_int< /*...*/ >& lhs,
                   const large_int< /*...*/ >& rhs);
    template<class T2, class U2>
    static bool eq(const large_int< /*...*/ >& lhs,
                   const large_int< /*...*/ >& rhs);
    static bool lt_literal(const large_int< /*...*/ >& lhs, long rhs);
    static bool eq_literal(const large_int< /*...*/ >& lhs, long rhs);
    static bool is_neg(const large_int< /*...*/ >& val);
    static bool is_zero(const large_int< /*...*/ >& val);
};

} // namespace large_int
} // namespace boost
```

`low_part_type` — The type of the respective `boost::large_int::large_int`'s low-part.

`high_part_type` — The type of the respective `boost::large_int::large_int`'s high-part.

`is_large_int` — For detection of `boost::large_int::large_int` types in generic programming. Will have the value `true` if the for all `boost::large_int::large_int` based types, `false` otherwise.

`low_bits` — The size of the respective `boost::large_int::large_int`'s low-part, as a number of binary bits.

`high_bits` — The size of the respective `boost::large_int::large_int`'s high-part, as a number of binary bits.

`size_bits` — The size of the respective `boost::large_int::large_int`, in total, as a number of binary bits.

`lt` — Compares two `boost::large_int::large_int` values, returning true if and only if the value of `lhs` is strictly less-than the value of `rhs`. (Where the types of `lhs` and `rhs` do not match, either `lhs` or `rhs` will be automatically promoted to the larger of the two integer types.)

`eq` — Compares two `boost::large_int::large_int` values, returning true if and only if the value of `lhs` is strictly equal-to the value of `rhs`. (Where the types of `lhs` and `rhs` do not match, either `lhs` or `rhs` will be automatically promoted to the larger of the two integer types.)

`lt_literal` — Compares a `boost::large_int::large_int` value to a literal, returning true if and only if the value of `lhs` is strictly less-than the value of `rhs`.

`eq_literal` — Compares a `boost::large_int::large_int` value to a literal, returning true if and only if the value of `lhs` is strictly equal-to the value of `rhs`.

`is_neg` — Returns true if and only if a given `boost::large_int::large_int` value is negative (less-than zero). This function may also be used upon unsigned `large_int` values (or where the signed status is unknown) without raising a warning, or error.

`is_zero` — Returns true if and only if a given `boost::large_int::large_int` value is equal-to zero. Calls to this function will typically be faster than an explicit comparison of a `large_int` value against either `large_int` or literal zero, as it is unary.

> **Warning**
>
> Where large_int_traits is used with a non-`boost::large_int::large_int` based type, only the boolean constant `is_large_int` will be defined (with the value: false).

Usage Example

```
#include <iostream> // For std::cout, std::endl
#include <boost/cstdint.hpp> // For uint64_t, int8_t
#include <boost/large_int.hpp> // For boost::large_int::large_int,
                               //     boost::large_int::large_int_traits
void example()
{
    // Print the size, in bits, of the high
    // and low parts of a 72bit large_int
    typedef boost::large_int::large_int<uint64_t, int8_t> lint72_t;
    typedef boost::large_int::large_int_traits<lint72_t> traits_type;
    std::cout << "lint72_t is composed of -"
              << traits_type::high_bits << " high-part + -"
              << traits_type::low_bits << " low-part bits"
              << std::endl;
}
```

Large Integer Utils

Miscellaneous utility functions for use with `boost::large_int::large_int` values.

abs

Like `std::abs`, `boost::large_int::abs` returns the absolute-value of a given `boost::large_int::large_int` type argument.

```
namespace boost {
namespace large_int {

template<class T, class U> large_int<T, U> abs(const large_int<T, U>& val);

} // namespace large_int
} // namespace boost
```

Usage Example

```
#include <iostream> // For std::cin, std::cout, std::flush, std::endl
#include <boost/cstdint.hpp> // For uint64_t, int8_t
#include <boost/large_int.hpp> // For boost::large_int::large_int,
                               //     boost::large_int::abs
#include <boost/large_int/io.hpp> // For streaming operators << and >>

void example()
{
    // Take a number from the command-line
    // and print its positive equivalent
    boost::large_int::large_int<uint64_t, int8_t> val;
    std::cout << "Input a number please: -" << std::flush;
    if( std::cin >> val )
    {
        std::cout << "abs(" << val << ") == -"
                  << boost::large_int::abs(val)
                  << std::endl;
    }
}
```

Caution

For both `boost::large_int::large_int` and built-in signed integer types, the minimum value as returned by `std::numeric_limits<T>::min()` will typically not have a usable positive equivalent for the same number of integer bits, and will therefore remain negative after any call to `abs`.

make_large_int

For the simple creation of large integer values, Boost.LargeInt also contains the function template `boost::large_int::make_large_int`. This function will attempt to parse and convert a given character sequence to its `large_int` equivalent.

```
namespace boost {
namespace large_int {

template<class T, class CharT> T make_large_int(const CharT* s);
template<class T, class InputIterator> T make_large_int(
    InputIterator first, InputIterator last, int base = 0);
template<class T, class InputIterator> InputIterator make_large_int(
    InputIterator first, InputIterator last, int base, T& result);

} // namespace large_int
} // namespace boost
```

Where:

`s` A pointer to the start of a C-string representing the number to be converted.

`first` An iterator to the start of a character-sequence representing the number to be converted.

`last` An iterator to one-past-the-end of a character-sequence representing the number to be converted.

`base` The default base of the input number, in the range `[0...MAX_INT]`. If zero, the base will be read from the character sequence as a standard prefix, where: "`0`" represents octal (base 8), "`0x`" represents hexadecimal (base 16), and no prefix represents decimal (base 10).

`result` The destination `boost::large_int::large_int` value.

The final version of this function (which returns an `InputIterator` result) will return the position of the first character which was not or could not be converted to a `large_int` value.

Usage Example

```
#include <string> // For std::string
#include <boost/cstdint.hpp> // For uint64_t, int8_t
#include <boost/large_int.hpp> // For boost::large_int::large_int,
                               //     boost::large_int::make_large_int
void example()
{
    // Declare a literal large_int value from a C-string
    typedef boost::large_int::large_int<uint64_t, int8_t> lint72_t;
    lint72_t literal(boost::large_int::make_large_int<lint72_t>(
        "0x0123456789abcdefAB"));

    // Replace it with a literal, octal large_int value from a std::string
    std::string literal_src("777");
    literal = boost::large_int::make_large_int<lint72_t>(
        literal_src.begin(), literal_src.end(), 8);
}
```

Large Integer Defaults

Standard `boost::large_int::large_int` based integral types. These integral types are provided in addition to those supplied by the headers: '`stdint.h`', '`cstdint`' or '`boost/cstdint.hpp`', and have similar names to those types for ease of identification and use.

Provided integer types:

Type Name	**Signed/Unsigned**	**Size**
`boost::large_int::lint64_t`	signed	64bit (32bit low-part, 32bit high-part)
`boost::large_int::luint64_t`	unsigned	64bit (32bit low-part, 32bit high-part)
`boost::large_int::lint96_t`	signed	96bit (64bit low-part, 32bit high-part)
`boost::large_int::luint96_t`	unsigned	96bit (64bit low-part, 32bit high-part)
`boost::large_int::lint128_t`	signed	128bit (64bit low-part, 64bit high-part)
`boost::large_int::luint128_t`	unsigned	128bit (64bit low-part, 64bit high-part)
`boost::large_int::lint160_t`	signed	160bit (128bit low-part, 32bit high-part)
`boost::large_int::luint160_t`	unsigned	160bit (128bit low-part, 32bit high-part)
`boost::large_int::lint192_t`	signed	192bit (128bit low-part, 64bit high-part)
`boost::large_int::luint192_t`	unsigned	192bit (128bit low-part, 64bit high-part)
`boost::large_int::lint256_t`	signed	256bit (128bit low-part, 128bit high-part)
`boost::large_int::luint256_t`	unsigned	256bit (128bit low-part, 128bit high-part)

Where the underlying platform has built-in support for any of the above integer types (most commonly for signed and unsigned 64bit), the available built-in types will be used in preference to large_int, for efficiency.

> **Tip**
>
> Where compiler support for 64bit integers is unknown or unavailable, the given lint64_t or luint64_t types could be used rather than testing for `BOOST_NO_INT64_T`, without any degradation to performance.

Large Integer I/O

To match the built-in integral types, Boost.LargeInt also provides operator templates for the streaming operators << and >>. These may be used in the same manner as their built-in equivalents.

```
namespace boost {
namespace large_int {

template<class CharT, class Traits, class T, class U>
std::basic_istream<CharT, Traits>& operator>> (
    std::basic_istream<CharT, Traits>& i,
    large_int<T, U>& val);
template<class CharT, class Traits, class T, class U>
std::basic_ostream<CharT, Traits>& operator<< (
    std::basic_ostream<CharT, Traits>& o,
    const large_int<T, U>& val);

} // namespace large_int
} // namespace boost
```

Due to the inherent compilation overhead of including the standard '`istream`', '`ostream`' or '`iostream`' headers, the implementations for these streaming operators is **not** included by the header '`boost/large_int.hpp`'. Where required, support for these operators must be explicitly requested via:

```
#include <boost/large_int/io.hpp> // For streaming operators << and >>
```

Usage Example

```
#include <iostream> // For std::cin, std::cout, std::cerr,
                    //     std::flush, std::endl
#include <boost/cstdint.hpp> // For uint64_t, int8_t
#include <boost/large_int.hpp> // For boost::large_int::large_int,
                               //     boost::large_int::abs
#include <boost/large_int/io.hpp> // For streaming operators << and >>

void example()
{
    // Take a number from the command-line and echo it back, unaltered
    boost::large_int::large_int<uint64_t, int8_t> val;
    std::cout << "Enter a number please: -" << std::flush;
    if( std::cin >> val )
    {
        std::cout << "You entered: -" << val << std::endl;
    }
    else
    {
        std::cerr << "That's not a number!" << std::endl;
    }
```

```
}
```

Large Integer Version

The version of the Boost.LargeInt library is available as the defined value: `BOOST_LARGE_INT_VERSION`. This is an integral representation of the standard "<MAJOR>.<MINOR>.<REVISION>" versioning system, and may be interpreted as shown below:

```
// Large_int library version identifier ---
// Revision      = BOOST_LARGE_INT_VERSION % 100
// Minor version = BOOST_LARGE_INT_VERSION -/ 100 % 1000
// Major version = BOOST_LARGE_INT_VERSION -/ 100000
#define BOOST_LARGE_INT_VERSION 100203 // -"1.2.3"
```

Usage Example

```
#include <iostream> // For std::cout, std::endl
#include <boost/large_int/version.hpp> // For BOOST_LARGE_INT_VERSION

void example()
{
    // Print the current version of Boost.LargeInt to stdout
    std::cout << "The currently installed version of Boost.LargeInt is: -"
              << (BOOST_LARGE_INT_VERSION / 100000) << '.'
              << (BOOST_LARGE_INT_VERSION / 100 % 1000) << '.'
              << (BOOST_LARGE_INT_VERSION % 100)
              << std::endl;
}
```