Boost.Integer

Beman Dawes
Daryle Walker
Gennaro Prota
John Maddock

Table of Contents

Overview

Boost.Integer provides integer type support, particularly helpful in generic programming. It provides standard C99 integer types, such as might be found in <stdint.h>, without requiring that header. It provides the means to select an integer type based upon its properties, like the number of bits or the maximum supported value, as well as compile-time bit mask selection. There is a derivative of std::numeric_limits that provides integral constant expressions for `min` and `max`. Finally, it provides two compile-time algorithms: determining the highest power of two in a compile-time value; and computing min and max of constant expressions.

Component	Header	Purpose
Forward Declarations.	`<boost/integer_fwd.hpp>`	Forward declarations of classes and class templates - for use when just the name of a class is needed.
Standard Integer Types.	`<boost/cstdint.hpp>`	Provides typedef's based on the 1999 C Standard header `<stdint.h>`, wrapped in namespace boost. This implementation may #include the compiler supplied `<stdint.h>`, if present.
Integer Traits.	`<boost/integer_traits.hpp>`	Class template `boost::integer_traits`, derives from `std::numeric_limits` and adds `const_min` and `const_max` members.
Integer Type Selection.	`<boost/integer.hpp>`	Templates for integer type selection based on properties such as maximum value or number of bits: Use to select the type of an integer when some property such as maximum value or number of bits is known. Useful for generic programming.
Integer Masks.	`<boost/integer/integer_mask.hpp>`	Templates for the selection of integer masks, single or lowest group, based on the number of bits: Use to select a particular mask when the bit position(s) are based on a compile-time variable. Useful for generic programming.
Compile time log2 Calculation.	`<boost/integer/static_log2.hpp>`	Template for finding the highest power of two in a number: Use to find the bit-size/range based on a maximum value. Useful for generic programming.
Compile time min/max calculation.	`<boost/integer/static_min_max.hpp>`	Templates for finding the extrema of two numbers: Use to find a bound based on a minimum or maximum value. Useful for generic programming.

Standard Integer Types

Overview

The header `<boost/cstdint.hpp>` provides the typedef's useful for writing portable code that requires certain integer widths. All typedef's are in namespace boost.

The specifications for these types are based on the ISO/IEC 9899:1999 C Language standard header <stdint.h>. The 64-bit types required by the C standard are *not required* in the boost header, and may not be supplied for all platforms/compilers, because `long long` is not [yet] included in the C++ standard.

See cstdint_test.cpp for a test program.

Rationale

The organization of the Boost.Integer headers and classes is designed to take advantage of <stdint.h> types from the 1999 C standard without causing undefined behavior in terms of the 1998 C++ standard. The header <boost/cstdint.hpp> makes the standard integer types safely available in namespace `boost` without placing any names in namespace `std`. The intension is to complement rather than compete with the C++ Standard Library. Should some future C++ standard include <stdint.h> and <cstdint>, then <boost/cstdint.hpp> will continue to function, but will become redundant and may be safely deprecated.

Because these are boost headers, their names conform to boost header naming conventions rather than C++ Standard Library header naming conventions.

Caveat emptor

As an implementation artifact, certain C <limits.h> macro names may possibly be visible to users of <boost/cstdint.hpp>. Don't use these macros; they are not part of any Boost-specified interface. Use `boost::integer_traits<>` or `std::numeric_limits<>` instead.

As another implementation artifact, certain C <stdint.h> typedef names may possibly be visible in the global namespace to users of <boost/cstdint.hpp>. Don't use these names, they are not part of any Boost-specified interface. Use the respective names in namespace `boost` instead.

Exact-width integer types

The typedef `int#_t`, with # replaced by the width, designates a signed integer type of exactly # bits; for example `int8_t` denotes an 8-bit signed integer type. Similarly, the typedef `uint#_t` designates an unsigned integer type of exactly # bits.

These types are optional. However, if a platform supports integer types with widths of 8, 16, 32, 64, or any combination thereof, then <boost/cstdint.hpp> does provide the corresponding typedefs.

The absence of int64_t and uint64_t is indicated by the macro `BOOST_NO_INT64_T`.

Minimum-width integer types

The typedef `int_least#_t`, with # replaced by the width, designates a signed integer type with a width of at least # bits, such that no signed integer type with lesser size has at least the specified width. Thus, `int_least32_t` denotes the smallest signed integer type with a width of at least 32 bits. Similarly, the typedef name `uint_least#_t` designates an unsigned integer type with a width of at least # bits, such that no unsigned integer type with lesser size has at least the specified width.

The following minimum-width integer types are provided for all platforms:

- `int_least8_t`
- `int_least16_t`

- `int_least32_t`
- `uint_least8_t`
- `uint_least16_t`
- `uint_least32_t`

The following types are available only if, after including <boost/cstdint.hpp>, the macro BOOST_NO_INT64_T is not defined:

- `int_least64_t`
- `uint_least64_t`

All other minimum-width integer types are optional.

Fastest minimum-width integer types

The typedef `int_fast#_t`, with # replaced by the width, designates the fastest signed integer type with a width of at least # bits. Similarly, the typedef name `uint_fast#_t` designates the fastest unsigned integer type with a width of at least # bits.

There is no guarantee that these types are fastest for all purposes. In any case, however, they satisfy the signedness and width requirements.

The following fastest minimum-width integer types are provided for all platforms:

- `int_fast8_t`
- `int_fast16_t`
- `int_fast32_t`
- `uint_fast8_t`
- `uint_fast16_t`
- `uint_fast32_t`

The following types are available only if, after including <boost/cstdint.hpp>, the macro BOOST_NO_INT64_T is not defined:

- `int_fast64_t`
- `uint_fast64_t`

All other fastest minimum-width integer types are optional.

Greatest-width integer types

The typedef `intmax_t` designates a signed integer type capable of representing any value of any signed integer type.

The typedef `uintmax_t` designates an unsigned integer type capable of representing any value of any unsigned integer type.

These types are provided for all platforms.

Integer Constant Macros

The following macros are always defined after inclusion of this header, these allow integer constants of at least the specified width to be declared: INT8_C, UINT8_C, INT16_C, UINT16_C, INT32_C, UINT32_C, INTMAX_C, UINTMAX_C.

The macros INT64_C and UINT64_C are also defined if the the macro BOOST_NO_INT64_T is not defined.

The C99 macro __STDC_CONSTANT_MACROS is also defined as an artifact of the implementation.

For example:

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

// Here the constant 0x1FFFFFFFF has the correct suffix applied:
static const boost::uint64_t c = INT64_C(0x1FFFFFFFF);
```

Integer Traits

Motivation

The C++ Standard Library <limits> header supplies a class template `numeric_limits<>` with specializations for each fundamental type.

For integer types, the interesting members of `std::numeric_limits<>` are:

```
static const bool is_specialized;      // Will be true for integer types.
static T min() throw();                // Smallest representable value.
static T max() throw();                // Largest representable value.
static const int digits;               // For integers, the number of value bits.
static const int digits10;            // The number of base 10 digits that can be represented.
static const bool is_signed;           // True if the type is signed.
static const bool is_integer;          // Will be true for all integer types.
```

For many uses, these are sufficient. But min() and max() are problematical because they are not constant expressions (std::5.19), yet some usages require constant expressions.

The template class `integer_traits` addresses this problem.

Synopsis

```
namespace boost {
  template<class T>
  class integer_traits : public std::numeric_limits<T>
  {
  public:
     static const bool is_integral = false;
     //
     // These members are defined only if T is a built-in
     // integal type:
     //
     static const T const_min = implementation-defined;
     static const T const_max = implementation-defined;
  };
}
```

Description

Template class `integer_traits` is derived from `std::numeric_limits`. The primary specialization adds the single `bool` member `is_integral` with the compile-time constant value `false`. However, for all integral types `T` (std::3.9.1/7 [basic.fundamental]), there are specializations provided with the following compile-time constants defined:

member	**type**	**value**
`is_integral`	bool	`true`
`const_min`	`T`	equivalent to `std::numeric_limits<T>::min()`
`const_max`	`T`	equivalent to `std::numeric_limits<T>::max()`

Note: The *is_integral* flag is provided, because a user-defined integer class should specialize `std::numeric_limits<>::is_integer = true`, while compile-time constants `const_min` and `const_max` are not provided for that user-defined class, unless boost::integer_traits is also specialized.

Test Program

The program `integer_traits_test.cpp` exercises the `integer_traits` class.

Acknowledgements

Beman Dawes, Ed Brey, Steve Cleary, and Nathan Myers discussed the integer traits idea on the boost mailing list in August 1999.

Integer Type Selection

The <boost/integer.hpp> type selection templates allow integer types to be selected based on desired characteristics such as number of bits or maximum value. This facility is particularly useful for solving generic programming problems.

Synopsis

```
namespace boost
{
  //  fast integers from least integers
  template<typename LeastInt>
  struct int_fast_t
  {
      typedef implementation-defined-type  type;
  };

  //  signed
  template<int Bits>
  struct int_t
  {
      /* Member exact may or may not be defined depending upon Bits */
      typedef implementation-defined-type  exact;
      typedef implementation-defined-type  least;
      typedef int_fast_t<least>::fast      fast;
  };

  //  unsigned
  template<int Bits>
  struct uint_t
  {
      /* Member exact may or may not be defined depending upon Bits */
      typedef implementation-defined-type  exact;
      typedef implementation-defined-type  least;
      typedef int_fast_t<least>::fast      fast;
  };

  //  signed
  template<long long MaxValue>
  struct int_max_value_t
  {
      typedef implementation-defined-type  least;
      typedef int_fast_t<least>::fast      fast;
  };

  template<long long MinValue>
  struct int_min_value_t
  {
      typedef implementation-defined-type  least;
      typedef int_fast_t<least>::fast      fast;
  };

  //  unsigned
  template<unsigned long long Value>
  struct uint_value_t
  {
      typedef implementation-defined-type  least;
      typedef int_fast_t<least>::fast      fast;
  };
} // namespace boost
```

Easiest-to-Manipulate Types

The `int_fast_t` class template maps its input type to the next-largest type that the processor can manipulate the easiest, or to itself if the input type is already an easy-to-manipulate type. For instance, processing a bunch of `char` objects may go faster if they were converted to `int` objects before processing. The input type, passed as the only template parameter, must be a built-in integral type, except `bool`. Unsigned integral types can be used, as well as signed integral types. The output type is given as the nested type `fast`.

Implementation Notes: By default, the output type is identical to the input type. Eventually, this code's implementation should be customized for each platform to give accurate mappings between the built-in types and the easiest-to-manipulate built-in types. Also, there is no guarantee that the output type actually is easier to manipulate than the input type.

Sized Types

The `int_t`, `uint_t`, `int_max_value_t`, `int_min_value_t`, and `uint_value_t` class templates find the most appropiate built-in integral type for the given template parameter. This type is given by the nested type `least`. The easiest-to-manipulate version of that type is given by the nested type `fast`. The following table describes each template's criteria.

Table 1. Criteria for the Sized Type Class Templates

Class Template	Template Parameter Mapping
`boost::int_t<N>::least`	The smallest, built-in, signed integral type with at least *N* bits, including the sign bit. The parameter should be a positive number. A compile-time error results if the parameter is larger than the number of bits in the largest integer type.
`boost::int_t<N>::fast`	The easiest-to-manipulate, built-in, signed integral type with at least *N* bits, including the sign bit. The parameter should be a positive number. A compile-time error results if the parameter is larger than the number of bits in the largest integer type.
`boost::int_t<N>::exact`	A built-in, signed integral type with exactly *N* bits, including the sign bit. The parameter should be a positive number. Note that the member *exact* is defined **only** if there exists a type with exactly *N* bits.
`boost::uint_t<N>::least`	The smallest, built-in, unsigned integral type with at least *N* bits. The parameter should be a positive number. A compile-time error results if the parameter is larger than the number of bits in the largest integer type.
`boost::uint_t<N>::fast`	The easiest-to-manipulate, built-in, unsigned integral type with at least *N* bits. The parameter should be a positive number. A compile-time error results if the parameter is larger than the number of bits in the largest integer type.
`boost::uint_t<N>::exact`	A built-in, unsigned integral type with exactly *N* bits. The parameter should be a positive number. A compile-time error results if the parameter is larger than the number of bits in the largest integer type. Note that the member *exact* is defined **only** if there exists a type with exactly N bits.
`boost::int_max_value_t<V>::last`	The smallest, built-in, signed integral type that can hold all the values in the inclusive range *0* - *V*. The parameter should be a positive number.
`boost::int_max_value_t<V>::fast`	The easiest-to-manipulate, built-in, signed integral type that can hold all the values in the inclusive range *0* - *V*. The parameter should be a positive number.
`boost::int_min_value_t<V>::least`	The smallest, built-in, signed integral type that can hold all the values in the inclusive range *V* - *0*. The parameter should be a negative number.
`boost::int_min_value_t<V>::fast`	The easiest-to-manipulate, built-in, signed integral type that can hold all the values in the inclusive range *V* - *0*. The parameter should be a negative number.
`boost::uint_value_t<V>::least`	The smallest, built-in, unsigned integral type that can hold all positive values up to and including *V*. The parameter should be a positive number.

Class Template	Template Parameter Mapping
`boost::uint_value_t<V>::fast`	The easiest-to-manipulate, built-in, unsigned integral type that can hold all positive values up to and including *V*. The parameter should be a positive number.

Example

```
#include <boost/integer.hpp>

//...

int main()
{
    boost::int_t<24>::least my_var;  // my_var has at least 24-bits
    //...
    // This one is guarenteed not to be truncated:
    boost::int_max_value_t<1000>::least my1000 = 1000;
    //...
    // This one is guarenteed not to be truncated, and as fast
    // to manipulate as possible, its size may be greater than
    // that of my1000:
    boost::int_max_value_t<1000>::fast my_fast1000 = 1000;
}
```

Demonstration Program

The program integer_test.cpp is a simplistic demonstration of the results from instantiating various examples of the sized type class templates.

Rationale

The rationale for the design of the templates in this header includes:

- Avoid recursion because of concern about C++'s limited guaranteed recursion depth (17).
- Avoid macros on general principles.
- Try to keep the design as simple as possible.

Alternative

If the number of bits required is known beforehand, it may be more appropriate to use the types supplied in <boost/cstdint.hpp>.

Credits

The author of most of the Boost integer type choosing templates is Beman Dawes. He gives thanks to Valentin Bonnard and Kevlin Henney for sharing their designs for similar templates. Daryle Walker designed the value-based sized templates.

Integer Masks

Overview

The class templates in <boost/integer/integer_mask.hpp> provide bit masks for a certain bit position or a contiguous-bit pack of a certain size. The types of the masking constants come from the integer type selection templates header.

Synopsis

```
#include <cstddef>  // for std::size_t

namespace boost
{

template <std::size_t Bit>
struct high_bit_mask_t
{
    typedef implementation-defined-type  least;
    typedef implementation-defined-type  fast;

    static const least       high_bit       = implementation-defined;
    static const fast        high_bit_fast  = implementation-defined;

    static const std::size_t bit_position   = Bit;
};

template <std::size_t Bits>
struct low_bits_mask_t
{
    typedef implementation-defined-type  least;
    typedef implementation-defined-type  fast;

    static const least       sig_bits       = implementation-defined;
    static const fast        sig_bits_fast  = implementation-defined;

    static const std::size_t bit_count      = Bits;
};

// Specializations for low_bits_mask_t exist for certain bit counts.

}  // namespace boost
```

Single Bit-Mask Class Template

The `boost::high_bit_mask_t` class template provides constants for bit masks representing the bit at a certain position. The masks are equivalent to the value 2^{Bit}, where `Bit` is the template parameter. The bit position must be a nonnegative number from zero to *Max*, where Max is one less than the number of bits supported by the largest unsigned built-in integral type. The following table describes the members of an instantiation of `high_bit_mask_t`.

Table 2. Members of the `boost::high_bit_mask_t` Class Template

Member	Meaning
`least`	The smallest, unsigned, built-in type that supports the given bit position.
`fast`	The easiest-to-manipulate analog of `least`.
`high_bit`	A `least` constant of the value 2^{Bit}.
`high_bit_fast`	A `fast` analog of `high_bit`.
`bit_position`	The value of the template parameter, in case its needed from a renamed instantiation of the class template.

Group Bit-Mask Class Template

The `boost::low_bits_mask_t` class template provides constants for bit masks equivalent to the value (2^{Bits} - 1), where `Bits` is the template parameter. The parameter `Bits` must be a non-negative integer from zero to *Max*, where Max is the number of bits supported by the largest, unsigned, built-in integral type. The following table describes the members of `low_bits_mask_t`.

Table 3. Members of the [^boost::low_bits_mask_t] Class Template

Member	Meaning
`least`	The smallest, unsigned built-in type that supports the given bit count.
`fast`	The easiest-to-manipulate analog of `least`.
`sig_bits`	A `least` constant of the desired bit-masking value.
`sig_bits_fast`	A `fast` analog of `sig_bits`.
`bit_count`	The value of the template parameter, in case its needed from a renamed instantiation of the class template.

Implementation Notes

When `Bits` is the exact size of a built-in unsigned type, the implementation has to change to prevent undefined behavior. Therefore, there are specializations of `low_bits_mask_t` at those bit counts.

Example

```
#include <boost/integer/integer_mask.hpp>

//...

int main()
{
    typedef boost::high_bit_mask_t<29>  mask1_type;
    typedef boost::low_bits_mask_t<15>  mask2_type;

    mask1_type::least  my_var1;
    mask2_type::fast   my_var2;
    //...

    my_var1 |= mask1_type::high_bit;
    my_var2 &= mask2_type::sig_bits_fast;

    //...
}
```

Demonstration Program

The program integer_mask_test.cpp is a simplistic demonstration of the results from instantiating various examples of the bit mask class templates.

Rationale

The class templates in this header are an extension of the integer type selection class templates. The new class templates provide the same sized types, but also convenient masks to use when extracting the highest or all the significant bits when the containing built-in type contains more bits. This prevents contamination of values by the higher, unused bits.

Credits

The author of the Boost bit mask class templates is Daryle Walker.

Compile Time log2 Calculation

The class template in <boost/integer/static_log2.hpp> determines the position of the highest bit in a given value. This facility is useful for solving generic programming problems.

Synopsis

```
namespace boost
{

  typedef implementation-defined static_log2_argument_type;
  typedef implementation-defined static_log2_result_type;

  template <static_log2_argument_type arg>
  struct static_log2
  {
    static const static_log2_result_type value = implementation-defined;
  };


  template < >
  struct static_log2< 0 >
  {
    // The logarithm of zero is undefined.
  };


}  // namespace boost
```

Usage

The `boost::static_log2` class template takes one template parameter, a value of type `static_log2_argument_type`. The template only defines one member, `value`, which gives the truncated, base-two logarithm of the template argument.

Since the logarithm of zero, for any base, is undefined, there is a specialization of `static_log2` for a template argument of zero. This specialization has no members, so an attempt to use the base-two logarithm of zero results in a compile-time error.

Note:

- `static_log2_argument_type` is an *unsigned integer type* (C++ standard, 3.9.1p3).
- `static_log2_result_type` is an *integer type* (C++ standard, 3.9.1p7).

Demonstration Program

The program static_log2_test.cpp is a simplistic demonstration of the results from instantiating various examples of the binary logarithm class template.

Rationale

The base-two (binary) logarithm, abbreviated lb, function is occasionally used to give order-estimates of computer algorithms. The truncated logarithm can be considered the highest power-of-two in a value, which corresponds to the value's highest set bit (for binary integers). Sometimes the highest-bit position could be used in generic programming, which requires the position to be available statically (*i.e.* at compile-time).

Credits

The original version of the Boost binary logarithm class template was written by Daryle Walker and then enhanced by Giovanni Bajo with support for compilers without partial template specialization. The current version was suggested, together with a reference implementation, by Vesa Karvonen. Gennaro Prota wrote the actual source file.

Compile time min/max calculation

The class templates in <boost/integer/static_min_max.hpp> provide a compile-time evaluation of the minimum or maximum of two integers. These facilities are useful for generic programming problems.

Synopsis

```
namespace boost
{

typedef implementation-defined static_min_max_signed_type;
typedef implementation-defined static_min_max_unsigned_type;

template <static_min_max_signed_type Value1, static_min_max_signed_type Value2 >
    struct static_signed_min;

template <static_min_max_signed_type Value1, static_min_max_signed_type Value2>
    struct static_signed_max;

template <static_min_max_unsigned_type Value1, static_min_max_unsigned_type Value2>
    struct static_unsigned_min;

template <static_min_max_unsigned_type Value1, static_min_max_unsigned_type Value2>
    struct static_unsigned_max;

}
```

Usage

The four class templates provide the combinations for finding the minimum or maximum of two `signed` or `unsigned` (`long`) parameters, *Value1* and *Value2*, at compile-time. Each template has a single static data member, `value`, which is set to the respective minimum or maximum of the template's parameters.

Example

```
#include <boost/integer/static_min_max.hpp>

template < unsigned long AddendSize1, unsigned long AddendSize2 >
class adder
{
public:
    static  unsigned long  const  addend1_size = AddendSize1;
    static  unsigned long  const  addend2_size = AddendSize2;
    static  unsigned long  const  sum_size = boost::static_unsigned_max<AddendSize1, Addend↵
Size2>::value + 1;

    typedef int  addend1_type[ addend1_size ];
    typedef int  addend2_type[ addend2_size ];
    typedef int  sum_type[ sum_size ];

    void  operator ()( addend1_type const &a1, addend2_type const &a2, sum_type &s ) const;
};

//...

int main()
{
    int const   a1[] = { 0, 4, 3 };  // 340
    int const   a2[] = { 9, 8 };     //  89
    int         s[ 4 ];
    adder<3,2>  obj;

    obj( a1, a2, s );  // 's' should be 429 or { 9, 2, 4, 0 }
    //...
}
```

Demonstration Program

The program static_min_max_test.cpp is a simplistic demonstration of various comparisons using the compile-time extrema class templates.

Rationale

Sometimes the minimum or maximum of several values needs to be found for later compile-time processing, *e.g.* for a bound for another class template.

Credits

The author of the Boost compile-time extrema class templates is Daryle Walker.

History

1.42.0

- Reverted Trunk to release branch state (i.e. a "known good state").
- Fixed issues: 653, 3084, 3177, 3180, 3568, 3657, 2134.
- Added long long support to `boost::static_log2`, `boost::static_signed_min`, `boost::static_signed_max`, `boost::static_unsigned_minboost::static_unsigned_max`, when available.
- The argument type and the result type of `boost::static_signed_min` etc are now typedef'd. Formerly, they were hardcoded as `unsigned long` and `int` respectively. Please, use the provided typedefs in new code (and update old code as soon as possible).

1.32.0

- The argument type and the result type of `boost::static_log2` are now typedef'd. Formerly, they were hardcoded as `unsigned long` and `int` respectively. Please, use the provided typedefs in new code (and update old code as soon as possible).