Boost.NumericConversion

Fernando Luis Cacciola Carballal

Table of Contents

Overview

The Boost Numeric Conversion library is a collection of tools to describe and perform conversions between values of different numeric types.

The library includes a special alternative for a subset of `std::numeric_limits<>`, the bounds<> traits class, which provides a consistent way to obtain the boundary values for the range of a numeric type.

It also includes a set of trait classes which describes the compile-time properties of a conversion from a source to a target numeric type. Both arithmetic and user-defined numeric types can be used.

A policy-based converter object which uses `conversion_traits` to select an optimized implementation is supplied. Such implementation uses an optimal range checking code suitable for the source/target combination.

- The converter's out-of-range behavior can be customized via an OverflowHandler policy.
- For floating-point to integral conversions, the rounding mode can be selected via the Float2IntRounder policy.
- A custom low-level conversion routine (for UDTs for instance) can be passed via a RawConverter policy.
- The optimized automatic range-checking logic can be overridden via a UserRangeChecker policy.

Definitions

Introduction

This section provides definitions of terms used in the Numeric Conversion library.

> **Notation** underlined text denotes terms defined in the C++ standard.
>
> **bold face** denotes terms defined here but not in the standard.

Types and Values

As defined by the C++ Object Model (§1.7) the storage or memory on which a C++ program runs is a contiguous sequence of bytes where each byte is a contiguous sequence of bits.

An object is a region of storage (§1.8) and has a type (§3.9).

A type is a discrete set of values.

An object of type `T` has an object representation which is the sequence of bytes stored in the object (§3.9/4)

An object of type `T` has a value representation which is the set of bits that determine the *value* of an object of that type (§3.9/4). For POD types (§3.9/10), this bitset is given by the object representation, but not all the bits in the storage need to participate in the value representation (except for character types): for example, some bits might be used for padding or there may be trap-bits.

The **typed value** that is held by an object is the value which is determined by its value representation.

An **abstract value** (untyped) is the conceptual information that is represented in a type (i.e. the number π).

The **intrinsic value** of an object is the binary value of the sequence of unsigned characters which form its object representation.

Abstract values can be **represented** in a given type.

To **represent** an abstract value `V` in a type `T` is to obtain a typed value `v` which corresponds to the abstract value `V`.

The operation is denoted using the `rep()` operator, as in: `v=rep(V)`. `v` is the **representation** of `V` in the type `T`.

For example, the abstract value π can be represented in the type `double` as the `double value M_PI` and in the type `int` as the `int value 3`

Conversely, *typed values* can be **abstracted**.

To **abstract** a typed value `v` of type `T` is to obtain the abstract value `V` whose representation in `T` is `v`.

The operation is denoted using the `abt()` operator, as in: `V=abt(v)`.

`V` is the **abstraction** of `v` of type `T`.

Abstraction is just an abstract operation (you can't do it); but it is defined nevertheless because it will be used to give the definitions in the rest of this document.

C++ Arithmetic Types

The C++ language defines fundamental types (§3.9.1). The following subsets of the fundamental types are intended to represent *numbers*:

signed integer types (§3.9.1/2): `{signed char, signed short int, signed int, signed long int}` Can be used to represent general integer numbers (both negative and positive).

unsigned integer types (§3.9.1/3): `{unsigned char, unsigned short int, unsigned int, unsigned long int}` Can be used to represent positive integer numbers with modulo-arithmetic.

floating-point types (§3.9.1/8): `{float,double,long double}` Can be used to represent real numbers.

integral or integer types (§3.9.1/7): `{{signed integers},{unsigned integers}, bool, char and wchar_t}`

arithmetic types (§3.9.1/8): `{{integer types},{floating types}}`

The integer types are required to have a *binary* value representation.

Additionally, the signed/unsigned integer types of the same base type (`short`, `int` or `long`) are required to have the same value representation, that is:

```
         int i = -3 ; // suppose value representation is: 10011 (sign bit + 4 magnitude bits)
unsigned int u =  i ; // u is required to have the same 10011 as its value representation.
```

In other words, the integer types signed/unsigned X use the same value representation but a different *interpretation* of it; that is, their *typed values* might differ.

Another consequence of this is that the range for signed X is always a smaller subset of the range of unsigned X, as required by §3.9.1/3.

Note

Always remember that unsigned types, unlike signed types, have modulo-arithmetic; that is, they do not overflow. This means that:

- Always be extra careful when mixing signed/unsigned types

- Use unsigned types only when you need modulo arithmetic or very very large numbers. Don't use unsigned types just because you intend to deal with positive values only (you can do this with signed types as well).

Numeric Types

This section introduces the following definitions intended to integrate arithmetic types with user-defined types which behave like numbers. Some definitions are purposely broad in order to include a vast variety of user-defined number types.

Within this library, the term *number* refers to an abstract numeric value.

A type is **numeric** if:

- It is an arithmetic type, or,
- It is a user-defined type which
  - Represents numeric abstract values (i.e. numbers).

- Can be converted (either implicitly or explicitly) to/from at least one arithmetic type.
- Has range (possibly unbounded) and precision (possibly dynamic or unlimited).
- Provides an specialization of `std::numeric_limits`.

A numeric type is **signed** if the abstract values it represent include negative numbers.

A numeric type is **unsigned** if the abstract values it represent exclude negative numbers.

A numeric type is **modulo** if it has modulo-arithmetic (does not overflow).

A numeric type is **integer** if the abstract values it represent are whole numbers.

A numeric type is **floating** if the abstract values it represent are real numbers.

An **arithmetic value** is the typed value of an arithmetic type

A **numeric value** is the typed value of a numeric type

These definitions simply generalize the standard notions of arithmetic types and values by introducing a superset called numeric. All arithmetic types and values are numeric types and values, but not vice versa, since user-defined numeric types are not arithmetic types.

The following examples clarify the differences between arithmetic and numeric types (and values):

```
// A numeric type which is not an arithmetic type (is user-defined)
// and which is intended to represent integer numbers (i.e., an 'integer' numeric type)
class MyInt
{
    MyInt ( long long v ) ;
    long long to_builtin();
} ;
namespace std {
template<> numeric_limits<MyInt> { ... } ;
}

// A 'floating' numeric type (double) which is also an arithmetic type (built-in),
// with a float numeric value.
double pi = M_PI ;

// A 'floating' numeric type with a whole numeric value.
// NOTE: numeric values are typed valued, hence, they are, for instance,
// integer or floating, despite the value itself being whole or including
// a fractional part.
double two = 2.0 ;

// An integer numeric type with an integer numeric value.
MyInt i(1234);
```

Range and Precision

Given a number set `N`, some of its elements are representable in a numeric type `T`.

The set of representable values of type `T`, or numeric set of `T`, is a set of numeric values whose elements are the representation of some subset of `N`.

For example, the interval of `int` values `[INT_MIN,INT_MAX]` is the set of representable values of type `int`, i.e. the `int` numeric set, and corresponds to the representation of the elements of the interval of abstract values `[abt(INT_MIN),abt(INT_MAX)]` from the integer numbers.

Similarly, the interval of `double` values `[-DBL_MAX,DBL_MAX]` is the `double` numeric set, which corresponds to the subset of the real numbers from `abt(-DBL_MAX)` to `abt(DBL_MAX)`.

Let **`next(x)`** denote the lowest numeric value greater than x.

Let **`prev(x)`** denote the highest numeric value lower then x.

Let **`v=prev(next(V))`** and **`v=next(prev(V))`** be identities that relate a numeric typed value `v` with a number `V`.

An ordered pair of numeric values `x,y` s.t. `x<y` are **consecutive** iff `next(x)==y`.

The abstract distance between consecutive numeric values is usually referred to as a Unit in the Last Place, or **ulp** for short. A ulp is a quantity whose abstract magnitude is relative to the numeric values it corresponds to: If the numeric set is not evenly distributed, that is, if the abstract distance between consecutive numeric values varies along the set -as is the case with the floating-point types-, the magnitude of 1ulp after the numeric value `x` might be (usually is) different from the magnitude of a 1ulp after the numeric value y for `x!=y`.

Since numbers are inherently ordered, a **numeric set** of type `T` is an ordered sequence of numeric values (of type `T`) of the form:

```
REP(T)={l,next(l),next(next(l)),...,prev(prev(h)),prev(h),h}
```

where `l` and `h` are respectively the lowest and highest values of type `T`, called the boundary values of type `T`.

A numeric set is discrete. It has a **size** which is the number of numeric values in the set, a **width** which is the abstract difference between the highest and lowest boundary values: `[abt(h)-abt(l)]`, and a **density** which is the relation between its size and width: `density=size/width`.

The integer types have density 1, which means that there are no unrepresentable integer numbers between `abt(l)` and `abt(h)` (i.e. there are no gaps). On the other hand, floating types have density much smaller than 1, which means that there are real numbers unrepresented between consecutive floating values (i.e. there are gaps).

The interval of abstract values `[abt(l),abt(h)]` is the range of the type `T`, denoted `R(T)`.

A range is a set of abstract values and not a set of numeric values. In other documents, such as the C++ standard, the word `range` is *sometimes* used as synonym for `numeric set`, that is, as the ordered sequence of numeric values from `l` to `h`. In this document, however, a range is an abstract interval which subtends the numeric set.

For example, the sequence `[-DBL_MAX,DBL_MAX]` is the numeric set of the type `double`, and the real interval `[abt(-DBL_MAX),abt(DBL_MAX)]` is its range.

Notice, for instance, that the range of a floating-point type is *continuous* unlike its numeric set.

This definition was chosen because:

- **(a)** The discrete set of numeric values is already given by the numeric set.
- **(b)** Abstract intervals are easier to compare and overlap since only boundary values need to be considered.

This definition allows for a concise definition of `subranged` as given in the last section.

The width of a numeric set, as defined, is exactly equivalent to the width of a range.

The **precision** of a type is given by the width or density of the numeric set.

For integer types, which have density 1, the precision is conceptually equivalent to the range and is determined by the number of bits used in the value representation: The higher the number of bits the bigger the size of the numeric set, the wider the range, and the higher the precision.

For floating types, which have density <<1, the precision is given not by the width of the range but by the density. In a typical implementation, the range is determined by the number of bits used in the exponent, and the precision by the number of bits used in the mantissa (giving the maximum number of significant digits that can be exactly represented). The higher the number of exponent bits the wider the range, while the higher the number of mantissa bits, the higher the precision.

Exact, Correctly Rounded and Out-Of-Range Representations

Given an abstract value `V` and a type `T` with its corresponding range `[abt(l),abt(h)]`:

If `V < abt(l)` or `V > abt(h)`, `V` is **not representable** (cannot be represented) in the type `T`, or, equivalently, it's representation in the type `T` is **out of range**, or **overflows**.

- If `V < abt(l)`, the **overflow is negative**.
- If `V > abt(h)`, the **overflow is positive**.

If `V >= abt(l)` and `V <= abt(h)`, `V` is **representable** (can be represented) in the type `T`, or, equivalently, its representation in the type `T` is **in range**, or **does not overflow**.

Notice that a numeric type, such as a C++ unsigned type, can define that any `V` does not overflow by always representing not `V` itself but the abstract value `U = [ V % (abt(h)+1) ]`, which is always in range.

Given an abstract value `V` represented in the type `T` as `v`, the **roundoff** error of the representation is the abstract difference: `(abt(v)-V)`.

Notice that a representation is an *operation*, hence, the roundoff error corresponds to the representation operation and not to the numeric value itself (i.e. numeric values do not have any error themselves)

- If the roundoff is 0, the representation is **exact**, and `V` is exactly representable in the type `T`.
- If the roundoff is not 0, the representation is **inexact**, and `V` is inexactly representable in the type `T`.

If a representation `v` in a type `T` -either exact or inexact-, is any of the adjacents of `V` in that type, that is, if `v==prev` or `v==next`, the representation is faithfully rounded. If the choice between `prev` and `next` matches a given **rounding direction**, it is **correctly rounded**.

All exact representations are correctly rounded, but not all inexact representations are. In particular, C++ requires numeric conversions (described below) and the result of arithmetic operations (not covered by this document) to be correctly rounded, but batch operations propagate roundoff, thus final results are usually incorrectly rounded, that is, the numeric value `r` which is the computed result is neither of the adjacents of the abstract value `R` which is the theoretical result.

Because a correctly rounded representation is always one of adjacents of the abstract value being represented, the roundoff is guaranteed to be at most 1ulp.

The following examples summarize the given definitions. Consider:

- A numeric type `Int` representing integer numbers with a *numeric set*: `{-2,-1,0,1,2}` and *range*: `[-2,2]`
- A numeric type `Cardinal` representing integer numbers with a *numeric set*: `{0,1,2,3,4,5,6,7,8,9}` and *range*: `[0,9]` (no modulo-arithmetic here)

- A numeric type `Real` representing real numbers with a *numeric set*: `{-2.0,-1.5,-1.0,-0.5,-0.0,+0.0,+0.5,+1.0,+1.5,+2.0}` and *range*: `[-2.0,+2.0]`
- A numeric type `Whole` representing real numbers with a *numeric set*: `{-2.0,-1.0,0.0,+1.0,+2.0}` and *range*: `[-2.0,+2.0]`

First, notice that the types `Real` and `Whole` both represent real numbers, have the same range, but different precision.

- The integer number `1` (an abstract value) can be exactly represented in any of these types.
- The integer number `-1` can be exactly represented in `Int`, `Real` and `Whole`, but cannot be represented in `Cardinal`, yielding negative overflow.
- The real number `1.5` can be exactly represented in `Real`, and inexactly represented in the other types.
- If `1.5` is represented as either `1` or `2` in any of the types (except `Real`), the representation is correctly rounded.
- If `0.5` is represented as `+1.5` in the type `Real`, it is incorrectly rounded.
- `(-2.0,-1.5)` are the `Real` adjacents of any real number in the interval `[-2.0,-1.5]`, yet there are no `Real` adjacents for `x < -2.0`, nor for `x > +2.0`.

Standard (numeric) Conversions

The C++ language defines Standard Conversions (§4) some of which are conversions between arithmetic types.

These are Integral promotions (§4.5), Integral conversions (§4.7), Floating point promotions (§4.6), Floating point conversions (§4.8) and Floating-integral conversions (§4.9).

In the sequel, integral and floating point promotions are called **arithmetic promotions**, and these plus integral, floating-point and floating-integral conversions are called **arithmetic conversions** (i.e, promotions are conversions).

Promotions, both Integral and Floating point, are *value-preserving*, which means that the typed value is not changed with the conversion.

In the sequel, consider a source typed value `s` of type `S`, the source abstract value `N=abt(s)`, a destination type `T`; and whenever possible, a result typed value `t` of type `T`.

Integer to integer conversions are always defined:

- If `T` is unsigned, the abstract value which is effectively represented is not `N` but `M=[ N % ( abt(h) + 1 ) ]`, where `h` is the highest unsigned typed value of type `T`.
- If `T` is signed and `N` is not directly representable, the result `t` is implementation-defined, which means that the C++ implementation is required to produce a value `t` even if it is totally unrelated to `s`.

Floating to Floating conversions are defined only if `N` is representable; if it is not, the conversion has undefined behavior.

- If `N` is exactly representable, `t` is required to be the exact representation.
- If `N` is inexactly representable, `t` is required to be one of the two adjacents, with an implementation-defined choice of rounding direction; that is, the conversion is required to be correctly rounded.

Floating to Integer conversions represent not `N` but `M=trunc(N)`, were `trunc()` is to truncate: i.e. to remove the fractional part, if any.

- If `M` is not representable in `T`, the conversion has undefined behavior (unless `T` is `bool`, see §4.12).

Integer to Floating conversions are always defined.

- If `N` is exactly representable, `t` is required to be the exact representation.

- If `N` is inexactly representable, `t` is required to be one of the two adjacents, with an implementation-defined choice of rounding direction; that is, the conversion is required to be correctly rounded.

Subranged Conversion Direction, Subtype and Supertype

Given a source type `S` and a destination type `T`, there is a **conversion direction** denoted: `S->T`.

For any two ranges the following *range relation* can be defined: A range `X` can be *entirely contained* in a range `Y`, in which case it is said that `X` is enclosed by `Y`.

> **Formally:** `R(S)` is enclosed by `R(T)` iif `(R(S) intersection R(T)) == R(S)`.

If the source type range, `R(S)`, is not enclosed in the target type range, `R(T)`; that is, if `(R(S) & R(T)) != R(S)`, the conversion direction is said to be **subranged**, which means that `R(S)` is not entirely contained in `R(T)` and therefore there is some portion of the source range which falls outside the target range. In other words, if a conversion direction `S->T` is subranged, there are values in `S` which cannot be represented in `T` because they are out of range. Notice that for `S->T`, the adjective subranged applies to `T`.

Examples:

Given the following numeric types all representing real numbers:

- `X` with numeric set `{-2.0,-1.0,0.0,+1.0,+2.0}` and range `[-2.0,+2.0]`
- `Y` with numeric set `{-2.0,-1.5,-1.0,-0.5,0.0,+0.5,+1.0,+1.5,+2.0}` and range `[-2.0,+2.0]`
- `Z` with numeric set `{-1.0,0.0,+1.0}` and range `[-1.0,+1.0]`

For:

(a) X->Y: `R(X) & R(Y) == R(X)`, then `X->Y` is not subranged. Thus, all values of type `X` are representable in the type `Y`.

(b) Y->X: `R(Y) & R(X) == R(Y)`, then `Y->X` is not subranged. Thus, all values of type `Y` are representable in the type `X`, but in this case, some values are *inexactly* representable (all the halves). (note: it is to permit this case that a range is an interval of abstract values and not an interval of typed values)

(b) X->Z: `R(X) & R(Z) != R(X)`, then `X->Z` is subranged. Thus, some values of type `X` are not representable in the type `Z`, they fall out of range `(-2.0 and +2.0)`.

It is possible that `R(S)` is not enclosed by `R(T)`, while neither is `R(T)` enclosed by `R(S)`; for example, `UNSIG=[0,255]` is not enclosed by `SIG=[-128,127]`; neither is `SIG` enclosed by `UNSIG`. This implies that is possible that a conversion direction is subranged both ways. This occurs when a mixture of signed/unsigned types are involved and indicates that in both directions there are values which can fall out of range.

Given the range relation (subranged or not) of a conversion direction `S->T`, it is possible to classify `S` and `T` as **supertype** and **subtype**: If the conversion is subranged, which means that `T` cannot represent all possible values of type `S`, `S` is the supertype and `T` the subtype; otherwise, `T` is the supertype and `S` the subtype.

For example:

> `R(float)=[-FLT_MAX,FLT_MAX]` and `R(double)=[-DBL_MAX,DBL_MAX]`

If `FLT_MAX < DBL_MAX`:

- `double->float` is subranged and `supertype=double`, `subtype=float`.
- `float->double` is not subranged and `supertype=double`, `subtype=float`.

Notice that while `double->float` is subranged, `float->double` is not, which yields the same supertype,subtype for both directions.

Now consider:

`R(int)=[INT_MIN,INT_MAX]` and `R(unsigned int)=[0,UINT_MAX]`

A C++ implementation is required to have `UINT_MAX > INT_MAX` (§3.9/3), so:

- 'int->unsigned' is subranged (negative values fall out of range) and `supertype=int`, `subtype=unsigned`.
- 'unsigned->int' is *also* subranged (high positive values fall out of range) and `supertype=unsigned`, `subtype=int`.

In this case, the conversion is subranged in both directions and the supertype,subtype pairs are not invariant (under inversion of direction). This indicates that none of the types can represent all the values of the other.

When the supertype is the same for both `S->T` and `T->S`, it is effectively indicating a type which can represent all the values of the subtype. Consequently, if a conversion `X->Y` is not subranged, but the opposite `(Y->X)` is, so that the supertype is always `Y`, it is said that the direction `X->Y` is **correctly rounded value preserving**, meaning that all such conversions are guaranteed to produce results in range and correctly rounded (even if inexact). For example, all integer to floating conversions are correctly rounded value preserving.

converter<> function object

Synopsis

```
namespace boost { namespace numeric {


    template<class T,
             class S,
             class Traits,          = conversion_traits<T,S>
             class OverflowHandler  = def_overflow_handler,
             class Float2IntRounder = Trunc< typename Traits::source_type >,
             class RawConverter     = raw_converter<Traits>,
             class UserRangeChecker = UseInternalRangeChecker
            >
    struct converter
    {
        typedef Traits traits ;

        typedef typename Traits::source_type   source_type   ;
        typedef typename Traits::argument_type argument_type ;
        typedef typename Traits::result_type   result_type   ;

        static result_type convert ( argument_type s ) ;

        result_type operator() ( argument_type s ) const ;

        // Internal member functions:

        static range_check_result out_of_range      ( argument_type s ) ;
        static void               validate_range    ( argument_type s ) ;
        static result_type        low_level_convert ( argument_type s ) ;
        static source_type        nearbyint         ( argument_type s ) ;

    } ;

} } // namespace numeric, boost
```

`boost::numeric::converter<>` is a Unary Function Object encapsulating the code to perform a numeric conversion with the direction and properties specified by the Traits template parameter. It can optionally take some policies which can be used to customize its behavior. The `Traits` parameter is not a policy but the parameter that defines the conversion.

Template parameters

`T`	The Numeric Type which is the *Target* of the conversion.
`S`	The Numeric Type which is the *Source* of the conversion.
`Traits`	This must be a conversion traits class with the interface of `boost::numeric::conversion_traits`
`OverflowHandler`	**Stateless Policy** called to administrate the result of the range checking. It is a **Function Object** which receives the result of `out_of_range()` and is called inside the `validate_range()` static member function exposed by the converter.
`Float2IntRounder`	**Stateless Policy** which specifies the rounding mode used for float to integral conversions. It supplies the `nearbyint()` static member function exposed by the converter.
`RawConverter`	**Stateless Policy** which is used to perform the actual conversion. It supplies the `low_level_convert()` static member function exposed by the converter.
`UserRangeChecker`	*Special and Optional* **Stateless Policy** which can be used to override the internal range checking logic. If given, supplies alternative code for the `out_of_range()` and `validate_range()` static member functions exposed by the converter.

Member functions

```
static result_type converter<>::convert ( argument_type s ) ; // throw
```

This static member function converts an rvalue of type `source_type` to an rvalue of type `target_type`.

If the conversion requires it, it performs a range checking before the conversion and passes the result of the check to the overflow handler policy (the default policy throws an exception if out-of-range is detected)

The implementation of this function is actually built from the policies and is basically as follows:

```
result_type converter<>::convert ( argument_type s )
{
    validate_range(s); // Implemented by the internal range checking logic
                       // (which also calls the OverflowHandler policy)
                       // or externally supplied by the UserRangeChecker policy.

    s = nearbyint(s); // Externally supplied by the Float2IntRounder policy.
                      // NOTE: This is actually called only for float to int conversions.

    return low_level_convert(s); // Externally supplied by the RawConverter policy.
}
```

`converter<>::operator() const` just calls `convert()`

```
static range_check_result numeric_converter<>::out_of_range ( argument_type s ) ;
```

This internal static member function determines if the value `s` can be represented by the target type without overflow.

It does not determine if the conversion is *exact*; that is, it does not detect *inexact* conversions, only *out-of-range* conversions (see the Definitions for further details).

The return value is of enum type `boost::numeric::range_check_result`

The actual code for the range checking logic is optimized for the combined properties of the source and target types. For example, a non-subranged conversion (i.e: `int->float`), requires no range checking, so `out_of_range()` returns `cInRange` directly. See the following table for more details.

If the user supplied a UserRangeChecker policy, is this policy which implements this function, so the implementation is user defined, although it is expected to perform the same conceptual check and return the appropriate result.

```
static void numeric_converter<>::validate_range ( argument_type s ) ; // no throw
```

This internal static member function calls out_of_range(s), and passes the result to the OverflowHandler policy class.

For those Target/Source combinations which don't require range checking, this is an empty inline function.

If the user supplied a UserRangeChecker policy, is this policy which implements this function, so the implementation is user defined, although it is expected to perform the same action as the default. In particular, it is expected to pass the result of the check to the overflow handler.

```
static result_type numeric_converter<>::low_level_convert ( argument_type s ) ;
```

This internal static member function performs the actual conversion.

This function is externally supplied by the RawConverter policy class.

```
static source_type converter<>::nearbyint ( argument_type s ) ;
```

This internal static member function, which is only used for `float` to `int` conversions, returns an *integer* value of *floating-point type* according to some rounding direction.

This function is externally supplied by the Float2IntRounder policy class which encapsulates the specific rounding mode.

Internal Member Functions

These static member functions build the actual conversion code used by `convert()`. The user does not have to call these if calling `convert()`, since `convert()` calls them infernally, but they can be called separately for specific needs.

Range Checking Logic

The following table summarizes the internal range checking logic performed for each combination of the properties of Source and Target.

LowestT/HighestT denotes the highest and lowest values of the Target type, respectively.

`S(n)` is short for `static_cast<S>(n)` (`S` denotes the Source type).

`NONE` indicates that for this case there is no range checking.

```
int_to_int    |--> sig_to_sig     |--> subranged     |--> ( s >= S(LowestT) ) && ( s <= ↵
S(HighestT) )
              |                   |--> not subranged |--> NONE
              |
              |--> unsig_to_unsig |--> subranged     |--> ( s >= S(LowestT) ) && ( s <= ↵
S(HighestT) )
              |                   |--> not subranged |--> NONE
              |
            |--> sig_to_unsig   |--> pos subranged     |--> ( s >= S(0) ) && ( s <= S(HighestT) )
              |                   |--> not pos subranged |--> ( s >= S(0) )
              |
              |--> unsig_to_sig   |--> subranged     |--> ( s <= S(HighestT) )
              |                   |--> not subranged |--> NONE


int_to_float   |--> NONE


float_to_int   |--> round_to_zero         |--> ( s >  S(LowestT)-S(1)   ) && ( s <  ↵
S(HighestT)+S(1)   )
               |--> round_to_even_nearest |--> ( s >= S(LowestT)-S(0.5) ) && ( s <  ↵
S(HighestT)+S(0.5) )
             |--> round_to_infinity     |--> ( s >  S(LowestT)-S(1)   ) && ( s <= S(HighestT) ↵
       )
               |--> round_to_neg_infinity |--> ( s >= S(LowestT)        ) && ( s <  ↵
S(HighestT)+S(1)   )


float_to_float |--> subranged     |--> ( s >= S(LowestT) ) && ( s <= S(HighestT) )
               |--> not subranged |--> NONE
```

Examples

```
#include <cassert>
#include <boost/numeric/conversion/converter.hpp>

int main() {

    typedef boost::numeric::converter<int,double> Double2Int ;

    int x = Double2Int::convert(2.0);
    assert ( x == 2 );

    int y = Double2Int()(3.14); // As a function object.
    assert ( y == 3 ) ; // The default rounding is trunc.

    try
    {
        double m = boost::numeric::bounds<double>::highest();
        int z = Double2Int::convert(m); // By default throws positive_overflow()
    }
    catch ( boost::numeric::positive_overflow const& )
    {
    }

    return 0;
}
```

Type Requirements and User-defined-types support

Type Requirements

Both arithmetic (built-in) and user-defined numeric types require proper specialization of `std::numeric_limits<>` (that is, with (in-class) integral constants).

The library uses `std::numeric_limits<T>::is_specialized` to detect whether the type is builtin or user defined, and `std::numeric_limits<T>::is_integer`, `std::numeric_limits<T>::is_signed` to detect whether the type is integer or floating point; and whether it is signed/unsigned.

The default `Float2IntRounder` policies uses unqualified calls to functions `floor()` and `ceil()`; but the standard functions are introduced in scope by a using directive:

```
using std::floor ; return floor(s);
```

Therefore, for builtin arithmetic types, the std functions will be used. User defined types should provide overloaded versions of these functions in order to use the default rounder policies. If these overloads are defined within a user namespace argument dependent lookup (ADL) should find them, but if your compiler has a weak ADL you might need to put these functions some place else or write your own rounder policy.

The default `Trunc<>` rounder policy needs to determine if the source value is positive or not, and for this it evaluates the expression `s < static_cast<S>(0)`. Therefore, user defined types require a visible `operator<` in order to use the `Trunc<>` policy (the default).

UDT's special semantics

Conversion Traits

If a User Defined Type is involved in a conversion, it is *assumed* that the UDT has wider range than any built-in type, and consequently the values of some `converter_traits<>` members are hardwired regardless of the reality. The following table summarizes this:

- `Target=`*UDT* and `Source=`*built-in*
  - `subranged=false`
  - `supertype=Target`
  - `subtype=Source`
- `Target=`*built-in* and `Source=`*UDT*
  - `subranged=true`
  - `supertype=Source`
  - `subtype=Target`
- `Target=`*UDT* and `Source=`*UDT*
  - `subranged=false`
  - `supertype=Target`
  - `subtype=Source`

The `Traits` member `udt_mixture` can be used to detect whether a UDT is involved and to infer the validity of the other members as shown above.

Range Checking

Because User Defined Numeric Types might have peculiar ranges (such as an unbounded range), this library does not attempt to supply a meaningful range checking logic when UDTs are involved in a conversion. Therefore, if either Target or Source are not built-in types, the bundled range checking of the `converter<>` function object is automatically disabled. However, it is possible to supply a user-defined range-checker. See Special Policies

Special Policies

There are two components of the `converter<>` class that might require special behavior if User Defined Numeric Types are involved: the Range Checking and the Raw Conversion.

When both Target and Source are built-in types, the converter class uses an internal range checking logic which is optimized and customized for the combined properties of the types.

However, this internal logic is disabled when either type is User Defined. In this case, the user can specify an *external* range checking policy which will be used in place of the internal code. See numeric_cast_traits for details on using UDTs with `numeric_cast`.

The converter class performs the actual conversion using a Raw Converter policy. The default raw converter simply performs a `static_cast<Target>(source)`.

However, if the a UDT is involved, the `static_cast` might not work. In this case, the user can implement and pass a different raw converter policy. See RawConverter policy for details.

UDTs with numeric_cast

In order to employ UDTs with `numeric_cast`, the user should define a `numeric_cast_traits` specialization on the UDT for each conversion. Here is an example of specializations for converting between the UDT and any other type:

```
namespace boost { namespace numeric {
template <typename Source>
struct numeric_cast_traits<UDT, Source>
{
    typedef conversion_traits<UDT, Source>      conv_traits;

    //! The following are required:
    typedef YourOverflowHandlerPolicy           overflow_policy;
    typedef YourRangeCheckerPolicy<conv_traits> range_checking_policy;
    typedef YourFloat2IntRounderPolicy<Source>  rounding_policy;
};
template <typename Target>
struct numeric_cast_traits<Target, UDT>
{
    typedef conversion_traits<Target, UDT>      conv_traits;

    //! The following are required:
    typedef YourOverflowHandlerPolicy           overflow_policy;
    typedef YourRangeCheckerPolicy<conv_traits> range_checking_policy;
    typedef YourFloat2IntRounderPolicy<UDT>     rounding_policy;
};
}}//namespace boost::numeric;
```

These specializations are already defined with default values for the built-in numeric types. It is possible to disable the generation of specializations for built-in types by defining `BOOST_NUMERIC_CONVERSION_RELAX_BUILT_IN_CAST_TRAITS`. For details on defining custom policies see Converter Policies.

Here is a full example of how to define a custom UDT for use with `numeric_cast`:

```
//! Define a simple custom number
struct Double
    :   boost::ordered_field_operators
        <
            Double
          , boost::ordered_field_operators2< Double, long double
          , boost::ordered_field_operators2< Double, double
          , boost::ordered_field_operators2< Double, float
          , boost::ordered_field_operators2< Double, int
          , boost::ordered_field_operators2< Double, unsigned int
          , boost::ordered_field_operators2< Double, long
          , boost::ordered_field_operators2< Double, unsigned long
          , boost::ordered_field_operators2< Double, long long
          , boost::ordered_field_operators2< Double, unsigned long long
          , boost::ordered_field_operators2< Double, char
          , boost::ordered_field_operators2< Double, unsigned char
          , boost::ordered_field_operators2< Double, short
          , boost::ordered_field_operators2< Double, unsigned short
        > > > > > > > > > > > > > >
{
    Double()
        : v(0)
    {}

    template <typename T>
    explicit Double( T v )
        : v(static_cast<double>(v))
    {}

    template <typename T>
    Double& operator= ( T t )
    {
        v = static_cast<double>(t);
        return *this;
    }

    bool operator < ( const Double& rhs ) const
    {
        return v < rhs.v;
    }

    template <typename T>
    bool operator < ( T rhs ) const
    {
        return v < static_cast<double>(rhs);
    }

    bool operator > ( const Double& rhs ) const
    {
        return v > rhs.v;
    }

    template <typename T>
    bool operator > ( T rhs ) const
    {
        return v > static_cast<double>(rhs);
    }

    bool operator ==( const Double& rhs ) const
    {
        return v == rhs.v;
    }
```

```
template <typename T>
bool operator == ( T rhs ) const
{
    return v == static_cast<double>(rhs);
}

bool operator !() const
{
    return v == 0;
}

Double operator -() const
{
    return Double(-v);
}

Double& operator +=( const Double& t )
{
    v += t.v;
    return *this;
}

template <typename T>
Double& operator +=( T t )
{
    v += static_cast<double>(t);
    return *this;
}

Double& operator -=( const Double& t )
{
    v -= t.v;
    return *this;
}

template <typename T>
Double& operator -=( T t )
{
    v -= static_cast<double>(t);
    return *this;
}

Double& operator *= ( const Double& factor )
{
    v *= factor.v;
    return *this;
}

template <typename T>
Double& operator *=( T t )
{
    v *= static_cast<double>(t);
    return *this;
}

Double& operator /= (const Double& divisor)
{
    v /= divisor.v;
    return *this;
}

template <typename T>
Double& operator /=( T t )
```

```
    {
         v /= static_cast<double>(t);
        return (*this);
    }

    double v;
};

//! Define numeric_limits for the custom type.
namespace std
{
    template<>
    class numeric_limits<Double> : public numeric_limits<double>
    {
    public:

        //! Limit our Double to a range of +/- 100.0
        static Double (min)()
        {
            return Double(1.e-2);
        }

        static Double (max)()
        {
            return Double(1.e2);
        }

        static Double epsilon()
        {
            return Double( std::numeric_limits<double>::epsilon() );
        }
    };
}

//! Define range checking and overflow policies.
namespace custom
{
    //! Define a custom range checker
    template<typename Traits, typename OverFlowHandler>
    struct range_checker
    {
        typedef typename Traits::argument_type argument_type ;
        typedef typename Traits::source_type S;
        typedef typename Traits::target_type T;

        //! Check range of integral types.
        static boost::numeric::range_check_result out_of_range( argument_type s )
        {
            using namespace boost::numeric;
            if( s > bounds<T>::highest() )
                return cPosOverflow;
            else if( s < bounds<T>::lowest() )
                return cNegOverflow;
            else
                return cInRange;
        }

        static void validate_range ( argument_type s )
        {
            BOOST_STATIC_ASSERT( std::numeric_limits<T>::is_bounded );
            OverFlowHandler()( out_of_range(s) );
        }
    };
```

```
    //! Overflow handler
    struct positive_overflow{};
    struct negative_overflow{};

    struct overflow_handler
    {
        void operator() ( boost::numeric::range_check_result r )
        {
            using namespace boost::numeric;
            if( r == cNegOverflow )
                throw negative_overflow() ;
            else if( r == cPosOverflow )
                throw positive_overflow() ;
        }
    };

    //! Define a rounding policy and specialize on the custom type.
    template<class S>
    struct Ceil : boost::numeric::Ceil<S>{};

    template<>
    struct Ceil<Double>
    {
      typedef Double source_type;

      typedef Double const& argument_type;

      static source_type nearbyint ( argument_type s )
      {
#if !defined(BOOST_NO_STDC_NAMESPACE)
          using std::ceil ;
#endif
          return Double( ceil(s.v) );
      }

      typedef boost::mpl::integral_c< std::float_round_style, std::round_toward_infin↵
ity> round_style;
    };

    //! Define a rounding policy and specialize on the custom type.
    template<class S>
    struct Trunc: boost::numeric::Trunc<S>{};

    template<>
    struct Trunc<Double>
    {
      typedef Double source_type;

      typedef Double const& argument_type;

      static source_type nearbyint ( argument_type s )
      {
#if !defined(BOOST_NO_STDC_NAMESPACE)
          using std::floor;
#endif
          return Double( floor(s.v) );
      }

     typedef boost::mpl::integral_c< std::float_round_style, std::round_toward_zero> round_style;
    };
}//namespace custom;
```

```
namespace boost { namespace numeric {

    //! Define the numeric_cast_traits specializations on the custom type.
    template <typename S>
    struct numeric_cast_traits<Double, S>
    {
        typedef custom::overflow_handler                          overflow_policy;
        typedef custom::range_checker
                <
                    boost::numeric::conversion_traits<Double, S>
                  , overflow_policy
                >                                                 range_checking_policy;
        typedef boost::numeric::Trunc<S>                          rounding_policy;
    };

    template <typename T>
    struct numeric_cast_traits<T, Double>
    {
        typedef custom::overflow_handler                          overflow_policy;
        typedef custom::range_checker
                <
                    boost::numeric::conversion_traits<T, Double>
                  , overflow_policy
                >                                                 range_checking_policy;
        typedef custom::Trunc<Double>                             rounding_policy;
    };

    //! Define the conversion from the custom type to built-in types and vice-versa.
    template<typename T>
    struct raw_converter< conversion_traits< T, Double > >
    {
        static T low_level_convert ( const Double& n )
        {
            return static_cast<T>( n.v );
        }
    };

    template<typename S>
    struct raw_converter< conversion_traits< Double, S > >
    {
        static Double low_level_convert ( const S& n )
        {
            return Double(n);
        }
    };
}}//namespace boost::numeric;
```

bounds<> traits class

Introduction

To determine the ranges of numeric types with `std::numeric_limits` [18.2.1], different syntax have to be used depending on numeric type. Specifically, `numeric_limits<T>::min()` for integral types returns the minimum finite value, whereas for floating point types it returns the minimum positive normalized value. The difference in semantics makes client code unnecessarily complex and error prone.

`boost::numeric::bounds<>` provides a consistent interface for retrieving the maximum finite value, the minimum finite value and the minimum positive normalized value (0 for integral types) for numeric types. The selection of implementation is performed at compile time, so there is no runtime overhead.

traits class bounds<N>

```
template<class N>
struct bounds
{
    static N lowest  () { return implementation_defined; }
    static N highest () { return implementation_defined; }
    static N smallest() { return implementation_defined; }
};
```

Members

`lowest()`

Returns the minimum finite value, equivalent to `numeric_limits<T>::min()` when `T` is an integral type, and to `-numeric_limits<T>::max()` when `T` is a floating point type.

`highest()`

Returns the maximum finite value, equivalent to `numeric_limits<T>::max()`.

`smallest()`

Returns the smallest positive normalized value for floating point types with denormalization, or returns 0 for integral types.

Examples

The following example demonstrates the use of `numeric::bounds<>` and the equivalent code using `numeric_limits`:

```
#include <iostream>

#include <boost/numeric/conversion/bounds.hpp>
#include <boost/limits.hpp>

int main() {

    std::cout << "numeric::bounds versus numeric_limits example.\n";

    std::cout << "The maximum value for float:\n";
    std::cout << boost::numeric::bounds<float>::highest() << "\n";
    std::cout << std::numeric_limits<float>::max() << "\n";

    std::cout << "The minimum value for float:\n";
    std::cout << boost::numeric::bounds<float>::lowest() << "\n";
    std::cout << -std::numeric_limits<float>::max() << "\n";

    std::cout << "The smallest positive value for float:\n";
    std::cout << boost::numeric::bounds<float>::smallest() << "\n";
    std::cout << std::numeric_limits<float>::min() << "\n";

    return 0;
}
```

conversion_traits<> traits class

Types

enumeration int_float_mixture_enum

```
namespace boost { namespace numeric {

    enum int_float_mixture_enum
    {
       integral_to_integral
      ,integral_to_float
      ,float_to_integral
      ,float_to_float
    } ;

} } // namespace boost::numeric
```

enumeration sign_mixture_enum

```
namespace boost { namespace numeric {

enum sign_mixture_enum
{
   unsigned_to_unsigned
  ,signed_to_signed
  ,signed_to_unsigned
  ,unsigned_to_signed
} ;

} } // namespace boost::numeric
```

enumeration udt_builtin_mixture_enum

```
namespace boost { namespace numeric {

    enum udt_builtin_mixture_enum
    {
       builtin_to_builtin
      ,builtin_to_udt
      ,udt_to_builtin
      ,udt_to_udt
    } ;

} } // namespace boost::numeric
```

template class int_float_mixture<>

```
namespace boost { namespace numeric {

    template <class T, class S>
    struct int_float_mixture : mpl::integral_c<int_float_mixture_enum, impl-def-value> {} ;

} } // namespace boost::numeric
```

Classifying `S` and `T` as either integral or float, this MPL's Integral Constant indicates the combination of these attributes.

Its `::value` is of enumeration type `boost::numeric::int_float_mixture_enum`

template class sign_mixture<>

```
namespace boost { namespace numeric {

    template <class T, class S>
    struct sign_mixture : mpl::integral_c<sign_mixture_enum, impl-def-value> {} ;

} } // namespace boost::numeric
```

Classifying `S` and `T` as either signed or unsigned, this MPL's Integral Constant indicates the combination of these attributes.

Its `::value` is of enumeration type `boost::numeric::sign_mixture_enum`

template class udt_builtin_mixture<>

```
namespace boost { namespace numeric {

    template <class T, class S>
   struct udt_builtin_mixture : mpl::integral_c<udt_builtin__mixture_enum, impl-def-value> {} ;

} } // namespace boost::numeric
```

Classifying `S` and `T` as either user-defined or builtin, this MPL's Integral Constant indicates the combination of these attributes.

Its `::value` is of enumeration type `boost::numeric::udt_builtin_mixture_enum`

template class is_subranged<>

```
namespace boost { namespace numeric {

    template <class T, class S>
    struct is_subranged : mpl::bool_<impl-def-value> {} ;

} } // namespace boost::numeric
```

Indicates if the range of the target type `T` is a subset of the range of the source type `S`. That is: if there are some source values which fall out of the Target type's range.

It is a boolean MPL's Integral Constant .

It does not indicate if a particular conversion is effectively out of range; it indicates that some conversion might be out of range because not all the source values are representable as Target type.

template class conversion_traits<>

```
namespace boost { namespace numeric {

    template <class T, class S>
    struct conversion_traits
    {
        mpl::integral_c<int_float_mixture_enum  , ...> int_float_mixture ;
        mpl::integral_c<sign_mixture_enum       , ...> sign_mixture;
        mpl::integral_c<udt_builtin_mixture_enum, ...> udt_builtin_mixture ;

        mpl::bool_<...> subranged ;
        mpl::bool_<...> trivial ;

        typedef T target_type   ;
        typedef S source_type   ;
        typedef ... argument_type ;
        typedef ... result_type   ;
        typedef ... supertype     ;
        typedef ... subtype       ;
    } ;

} } // namespace numeric, namespace boost
```

This traits class indicates some properties of a *numeric conversion* direction: from a source type `S` to a target type `T`. It does not indicate the properties of a *specific* conversion, but of the conversion direction. See Definitions for details.

The traits class provides the following MPL's Integral Constant \s of enumeration type. They express the combination of certain attributes of the Source and Target types (thus they are call mixture):

int_float_mixture	Same as given by the traits class int_float_mixture
sign_mixture	Same as given by the traits class sign_mixture
udt_builtin_mixture	Same as given by the traits class udt_builtin_mixture

The traits class provides the following MPL's Integral Constant \s of boolean type which indicates indirectly the relation between the Source and Target ranges (see Definitions for details).

subranged	Same as given by is_subranged
trivial	Indicates if both Source and Target, without cv-qualifications, are the same type. Its `::value` is of boolean type.

The traits class provides the following types. They are the Source and Target types classified and qualified for different purposes.

target_type	The template parameter `T` without cv-qualifications
source_type	The template parameter `S` without cv-qualifications
argument_type	This type is either source_type or `source_type const&`. It represents the optimal argument type for the converter member functions. If S is a built-in type, this is `source_type`, otherwise, this is `source_type const&`.
result_type	This type is either target_type or target_type const& It represents the return type of the converter member functions. If `T==S`, it is `target_type const&`, otherwise, it is `target_type`.
supertype	If the conversion is subranged, it is `source_type`, otherwise, it is `target_type`
subtype	If the conversion is subranged, it is `target_type`, otherwise, it is `source_type`

Examples

```
#include <cassert>
#include <typeinfo>
#include <boost/numeric/conversion/conversion_traits.hpp>

int main()
{

    // A trivial conversion.
    typedef boost::numeric::conversion_traits<short,short> Short2Short_Traits ;
    assert ( Short2Short_Traits::trivial::value ) ;

    // A subranged conversion.
    typedef boost::numeric::conversion_traits<double,unsigned int> UInt2Double_Traits ;
   assert (  UInt2Double_Traits::int_float_mixture::value == boost::numeric::integral_to_float ) ;
    assert (  UInt2Double_Traits::sign_mixture::value == boost::numeric::unsigned_to_signed ) ;
    assert ( !UInt2Double_Traits::subranged::value ) ;
    assert ( typeid(UInt2Double_Traits::supertype) == typeid(double) ) ;
    assert ( typeid(UInt2Double_Traits::subtype) == typeid(unsigned int) ) ;

    // A doubly subranged conversion.
    assert ( (boost::numeric::conversion_traits<short, unsigned short>::subranged::value) );
    assert ( (boost::numeric::conversion_traits<unsigned short, short>::subranged::value) );

    return 0;
}
```

Numeric Converter Policy Classes

enum range_check_result

```
namespace boost { namespace numeric {

    enum range_check_result
    {
        cInRange     ,
        cNegOverflow ,
        cPosOverflow
    } ;

} }
```

Defines the values returned by `boost::numeric::converter<>::out_of_range()`

Policy OverflowHandler

This *stateless* non-template policy class must be a *function object* and is called to administrate the result of the range checking. It can throw an exception if overflow has been detected by the range checking as indicated by its argument. If it throws, is is recommended that it be `std::bad_cast` or derived.

It must have the following interface (it does not has to be a template class):

```
struct YourOverflowHandlerPolicy
{
    void operator() ( boost::range_check_result ) ; // throw bad_cast or derived
} ;
```

It is called with the result of the converter's `out_of_range()` inside `validate_range()`.

These are the two overflow handler classes provided by the library:

```
namespace boost { namespace numeric {

    struct def_overflow_handler
    {
        void operator() ( range_check_result r ) // throw bad_numeric_conversion derived
        {
            if ( r == cNegOverflow )
                throw negative_overflow() ;
            else if ( r == cPosOverflow )
                throw positive_overflow() ;
        }
    } ;

    struct silent_overflow_handler
    {
        void operator() ( range_check_result ) // no-throw
        {}
    } ;

} }
```

And these are the Exception Classes thrown by the default overflow handler (see IMPORTANT note)

```
namespace boost { namespace numeric {


    class bad_numeric_cast : public std::bad_cast
    {
        public:
        virtual const char *what() const // throw()
        {
            return "bad numeric conversion: overflow";
        }

    };


    class negative_overflow : public bad_numeric_cast
    {
        public:
        virtual const char *what() const // throw()
        {
            return "bad numeric conversion: negative overflow";
        }
    };


    class positive_overflow : public bad_numeric_cast
    {
        public:
        virtual const char *what() const // throw()
        {
            return "bad numeric conversion: positive overflow";
        }
    };


} }
```

Important

RELEASE NOTE for 1.33 Previous to boost version 1.33, the exception class `bad_numeric_cast` was named `bad_numeric_conversion`. However, in 1.33, the old function `numeric_cast<>` from `boost/cast.hpp` was completly replaced by the new `numeric_cast<>` in `boost/numeric/conversion/cast.hpp` (and `boost/cast.hpp` is including `boost/numeric/conversion/cast.hpp` now). That old function which existed in boost for quite some time used the `bad_numeric_cast` as its exception type so I decided to avoid backward compatibility problems by adopting it (guessing that the user base for the old code is wider than for the new code).

Policy Float2IntRounder

This *stateless* template policy class specifies the rounding mode used for float to integral conversions. It supplies the `nearbyint()` static member function exposed by the converter, which means that it publicly inherits from this policy.

The policy must have the following interface:

```
template<class S>
struct YourFloat2IntRounderPolicy
{
    typedef S               source_type ;
    typedef {S or S const&} argument_type ;

    static source_type nearbyint ( argument_type s ) { ... }

    typedef mpl::integral_c<std::float_round_style,std::round_...> round_style ;

} ;
```

These are the rounder classes provided by the library (only the specific parts are shown, see the general policy form above)

Note

These classes are not intended to be general purpose rounding functions but specific policies for `converter<>`. This is why they are not function objects.

```
namespace boost { namespace numeric {


    template<class S>
    struct Trunc
    {
        static source_type nearbyint ( argument_type s )
        {
            using std::floor ;
            using std::ceil  ;

            return s >= static_cast<S>(0) ? floor(s) : ceil(s) ;
        }

        typedef mpl::integral_c<std::float_round_style,std::round_toward_zero> round_style ;
    } ;


    template<class S>
    struct RoundEven
    {
        static source_type nearbyint ( argument_type s )
        {
            return impl-defined-value ;
        }

        typedef mpl::integral_c<std::float_round_style,std::round_to_nearest> round_style ;
    } ;


    template<class S>
    struct Ceil
    {
        static source_type nearbyint ( argument_type s )
        {
            using std::ceil ;
            return ceil(s) ;
        }

       typedef mpl::integral_c<std::float_round_style,std::round_toward_infinity> round_style ;
    } ;
```

```
    template<class S>
    struct Floor
    {
        static source_type nearbyint ( argument_type s )
        {
            using std::floor ;
            return floor(s) ;
        }
        typedef mpl::integral_c<std::float_round_style,std::round_toward_neg_infin↵
ity> round_style ;
    } ;

} } // namespace numeric, namespace boost
```

Math Functions used by the rounder policies

The rounder policies supplied by this header use math functions `floor()` and `ceil()`. The standard versions of these functions are introduced in context by a using directive, so in normal conditions, the standard functions will be used.

However, if there are other visible corresponding overloads an ambiguity could arise. In this case, the user can supply her own rounder policy which could, for instance, use a fully qualified call.

This technique allows the default rounder policies to be used directly with user defined types. The user only requires that suitable overloads of `floor()` and `ceil()` be visible. See also User Defined Numeric Types support.

Policy RawConverter

This *stateless* template policy class is used to perform the actual conversion from Source to Target. It supplies the `low_level_convert()` static member function exposed by the converter, which means that it publicly inherits from this policy.

The policy must have the following interface:

```
template<class Traits>
struct YourRawConverterPolicy
{
    typedef typename Traits::result_type   result_type   ;
    typedef typename Traits::argument_type argument_type ;

    static result_type low_level_convert ( argument_type s ) { return <impl defined> ; }
} ;
```

This policy is mostly provided as a hook for user defined types which don't support `static_cast<>` conversions to some types

This is the only raw converter policy class provided by the library:

```
namespace boost { namespace numeric {

    template<class Traits>
    struct raw_numeric_converter
    {
        typedef typename Traits::result_type   result_type   ;
        typedef typename Traits::argument_type argument_type ;

        static result_type low_level_convert ( argument_type s )
        {
            return static_cast<result_type>(s) ;
        }
    } ;

} }
```

Policy UserRangeChecker

This *stateless* template policy class is used only if supplied to **override** the internal range checking logic.

It supplies the `validate_range()` static member function exposed by the converter, which means that it publicly inherits from this policy.

The policy must have the following interface:

```
template<class Traits>
struct YourRangeCheckerPolicy
{
    typedef typename Traits::argument_type argument_type ;

    // Determines if the value 's' fits in the range of the Target type.
    static range_check_result out_of_range ( argument_type s ) ;

    // Checks whether the value 's' is out_of_range()
    // and passes the result of the check to the OverflowHandler policy.
    static void validate_range ( argument_type s )
    {
        OverflowHandler()( out_of_range(s) ) ;
    }
} ;
```

This policy is **only** provided as a hook for user defined types which require range checking (which is disabled by default when a UDT is involved). The library provides a class: `UseInternalRangeChecker{};` which is a *fake* `RangeChecker` policy used to signal the converter to use its internal range checking implementation.

Improved numeric_cast<>

Introduction

The lack of preservation of range makes conversions between numeric types error prone. This is true for both implicit conversions and explicit conversions (through `static_cast`). `numeric_cast` detects loss of range when a numeric type is converted, and throws an exception if the range cannot be preserved.

There are several situations where conversions are unsafe:

- Conversions from an integral type with a wider range than the target integral type.
- Conversions from unsigned to signed (and vice versa) integral types.
- Conversions from floating point types to integral types.

The C++ Standard does not specify the behavior when a numeric type is assigned a value that cannot be represented by the type, except for unsigned integral types [3.9.1.4], which must obey the laws of arithmetic modulo 2n (this implies that the result will be reduced modulo the number that is one greater than the largest value that can be represented). The fact that the behavior for overflow is undefined for all conversions (except the aforementioned unsigned to unsigned) makes any code that may produce positive or negative overflows exposed to portability issues.

By default `numeric_cast` adheres to the rules for implicit conversions mandated by the C++ Standard, such as truncating floating point types when converting to integral types. The implementation must guarantee that for a conversion to a type that can hold all possible values of the source type, there will be no runtime overhead.

numeric_cast

```
template <typename Target, typename Source> inline
Target numeric_cast( Source arg )
{
    typedef conversion_traits<Target, Source>   conv_traits;
    typedef numeric_cast_traits<Target, Source> cast_traits;
    typedef converter
        <
            Target,
            Source,
            conv_traits,
            typename cast_traits::overflow_policy,
            typename cast_traits::rounding_policy,
            raw_converter<conv_traits>,
            typename cast_traits::range_checking_policy
        > converter;
    return converter::convert(arg);
}
```

`numeric_cast` returns the result of converting a value of type Source to a value of type Target. If out-of-range is detected, an overflow policy is executed whose default behavior is to throw an an exception (see bad_numeric_cast, negative_overflow and positive_overflow).

numeric_cast_traits

```
template <typename Target, typename Source, typename EnableIf = void>
struct numeric_cast_traits
{
    typedef def_overflow_handler    overflow_policy;
    typedef UseInternalRangeChecker range_checking_policy;
    typedef Trunc<Source>           rounding_policy;
};
```

The behavior of `numeric_cast` may be tailored for custom numeric types through the specialization of `numeric_cast_traits`. (see User Defined Types for details.)

Examples

The following example performs some typical conversions between numeric types:

1. include <boost/numeric/conversion/cast.hpp>

2. include <iostream>

```
int main()
{
    using boost::numeric_cast;

    using boost::numeric::bad_numeric_cast;
    using boost::numeric::positive_overflow;
    using boost::numeric::negative_overflow;

    try
    {
        int i=42;
        short s=numeric_cast<short>(i); // This conversion succeeds (is in range)
    }
    catch(negative_overflow& e) {
        std::cout << e.what();
    }
    catch(positive_overflow& e) {
        std::cout << e.what();
    }

    try
    {
        float f=-42.1234;

        // This will cause a boost::numeric::negative_overflow exception to be thrown
        unsigned int i=numeric_cast<unsigned int>(f);
    }
    catch(bad_numeric_cast& e) {
        std::cout << e.what();
    }

    double d= f + numeric_cast<double>(123); // int -> double

    unsigned long l=std::numeric_limits<unsigned long>::max();

    try
    {
        // This will cause a boost::numeric::positive_overflow exception to be thrown
        // NOTE: *operations* on unsigned integral types cannot cause overflow
        // but *conversions* to a signed type ARE range checked by numeric_cast.

        unsigned char c=numeric_cast<unsigned char>(l);
    }
    catch(positive_overflow& e) {
        std::cout << e.what();
    }


    return 0;
}
```

History and Acknowledgments

Pre-formal review

- Kevlin Henney, with help from David Abrahams and Beman Dawes, originally contributed the previous version of `numeric_cast<>` which already presented the idea of a runtime range check.
- Later, Eric Ford, Kevin Lynch and the author spotted some genericity problems with that `numeric_cast<>` which prevented it from being used in a generic layer of math functions.
- An improved `numeric_cast<>` which properly handled all combinations of arithmetic types was presented.
- David Abrahams and Beman Dawes acknowledged the need of an improved version of `numeric_cast<>` and supported the submission as originally laid out. Daryl Walker and Darin Adler made some important comments and proposed fixes to the original submission.
- Special thanks go to Björn Karlsoon who helped the author considerably. Having found the problems with `numeric_cast<>` himself, he revised very carefully the original submission and spot a subtle bug in the range checking implementation. He also wrote part of this documentation and proof-read and corrected other parts. And most importantly: the features now presented here in this library evolved from the original submission as a result of the useful private communications between Björn and the author.

Post-formal review

- Guillaume Melquiond spoted some documentation and code issues, particularly about rounding conversions.
- The following people contributed an important review of the design, documentation and c ode: Kevin Lynch, Thorsten Ottosen, Paul Bristow, Daryle Walker, Jhon Torjo, Eric Ford, Gennadiy Rozental.

Bibliography

- Standard Documents:
  1. ISO/IEC 14882:98 (C++98 Standard)
  2. ISO/IEC 9899:1999 (C99 Standard)
  3. ISO/IEC 10967-1 (Language Independent Arithmetic (LIA), Part I, 1994)
  4. ISO/IEC 2382-1:1993 (Information Technology - Vocabulary - Part I: Fundamental Terms)
  5. ANSI/IEEE 754-1985 [and IEC 60559:1989] (Binary floating-point)
  6. ANSI/IEEE 854-1988 (Radix Independent floating-point)
  7. ANSI X3/TR-1-82 (Dictionary for Information Processing Systems)
  8. ISO/IEC JTC1/SC22/WG14/N753 C9X Revision Proposal: LIA-1 Binding: Rationale
- Papers:
  1. David Goldberg What Every Computer Scientist Should Know About Floating-Point Arithmetic
  2. Prof. William Kahan papers on floating-point.