C++ Numerics Work In Progress Issues

Note: this is an early draft. It’s known to be incomplet and incorrekt, and it has lots of bad formatting.
Contents

1 Foundations .................................................. 1
  1.1 Mission .............................................. 1
  1.2 Style of presentation .................................. 1
  1.3 Scope ............................................. 1
  1.4 Design principles ................................... 1
  1.5 Number type taxonomy ................................ 2

2 Operations .................................................. 3

3 Machine Abstraction Layer .................................... 4
  3.1 Overflow-Detecting Operations ......................... 4
  3.2 Overflow-Handling Operations .......................... 4
  3.3 Rounding Operations .................................. 6
  3.4 Combined Rounding and Overflow Operations .......... 7
  3.5 Double-Word Operations ................................ 7

4 Machine extension layer .................................... 11
  4.1 Word-array operations ................................ 11

5 Feature test macros (Informative) .......................... 13

6 Wide Integers ................................................ 14
  6.1 Class template numeric_limits ......................... 14
  6.2 Header <wide_integer> synopsis ....................... 14
  6.3 Template class wide_integer overview ................. 17
  6.4 Specializations of common_type ...................... 20
  6.5 Unary operators .................................... 21
  6.6 Binary operators ................................... 21
  6.7 Numeric conversions ................................ 23
  6.8 iostream specializations ............................ 23
  6.9 Hash support ...................................... 23

7 Rational math ............................................... 24
  7.1 Class rational ...................................... 24

8 Parametric aliases ......................................... 30
  8.1 Freestanding implementations .......................... 30
  8.2 General ........................................... 30
  8.3 Header <cstdint> .................................. 30
  8.4 Parametric types .................................... 31
  8.5 Floating-point types ................................. 31
  8.6 Header <cstdfloat> ................................ 32
  8.7 Parametric types .................................... 32

9 Unbounded Types ............................................ 34
  9.1 Class integer_data_proxy ............................. 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.2</td>
<td>Bits</td>
<td>37</td>
</tr>
<tr>
<td>9.3</td>
<td>Integer</td>
<td>43</td>
</tr>
<tr>
<td>10</td>
<td>Generalized Type Conversion</td>
<td>53</td>
</tr>
</tbody>
</table>
1 Foundations

1.1 Mission
The mission of this document is to highlight inconsistencies among different numbers related proposals, provide a possible solution for them, and update the wordings to be more suitable for the LWG.

1.2 Style of presentation
Existing wording from the C++ working draft - included to provide context - is presented without decoration. Entire clauses / subclauses / paragraphs incorporated from different numbers related proposals are presented in a distinct cyan color. Wording to be added which is original to this document appears in blue. Wording which this document strikes is presented in red with strike-through. Notes and motivations for change are in brown.

1.3 Scope
This document provides numbers and their options, but does not provide algorithms or special functions.

1.4 Design principles
Follow mathematical behavior where feasible.

— Make unbounded numbers very simple to use.
— Make unavoidable overflow and rounding predictable and controllable.
  — Overflow on any operation is unmanageable.
  — A rational number with fixed-size fields needs to round.
— Avoid or hide aliasing effects.
— Prefer compile-time errors to run-time errors.
— Prefer safe defaults to efficient defaults.

Provide types that match the taxonomic needs.

— E.g. fixed-point, extended floating-point.
— Support construction of new types by library authors.
  — Expose common implementation abstractions.

Strive for efficiency.

— Use efficient function parameter passing.
— Use efficient representations.
— Give speed priority to dynamically common operations.
— Provide composite operations when efficient.
Ease adoption and use.

— Provide a consistent vocabulary.

— Enable value conversion between all applicable types.

— Handle parameter aliasing within the implementations.

Ease extension.

— Provide a mechanism for conversion that does not require $n^2$ operations or coordination between independent developers.

— Most parts of the implementation should need only C++, so provide a machine abstraction layer.

— Expose sound "building-block" abstractions.

1.5 Number type taxonomy

Types may be categorized by the representation constancy between argument and result.

same The size of the type is the same between argument and result. Overflow is pervasive on all operations.

— invariant - All aspects of the representation are the same.

— invariant - Different fields within the representation may have different sizes.

adaptive The size of the result is statically known, but it may be different from (and generally larger than) the argument.

— limited - There is a maximum representation. If adaptation requires more than the maximum, the expression is ill-formed. Overflow in expressions is avoided until the maximum representation is reached.

— unlimited - There is no a priori maximum representation. Overflow in expressions is avoided.

dynamic All aspects of the representation are dynamic. Overflow is variables is avoided.

Types may also be categorized by the size specification, which is generally by the number of bits in the representation or by the number of digits required.

Should we place all the new sections from this document into the [numerics] and adjust the tags accordingly?
2 Operations

The various types will share many operations. To reduce surprise, these operations should have consistent names for consistent purposes.

Table 1 — Operations

<table>
<thead>
<tr>
<th>operator</th>
<th>name</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>neg</td>
<td>negate</td>
</tr>
<tr>
<td>+</td>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>-</td>
<td>sub</td>
<td>subtract</td>
</tr>
<tr>
<td>*</td>
<td>mul</td>
<td>multiply</td>
</tr>
<tr>
<td></td>
<td>rdiv</td>
<td>divide with rounding</td>
</tr>
<tr>
<td></td>
<td>pdiv</td>
<td>divide with promotion</td>
</tr>
<tr>
<td></td>
<td>quorem</td>
<td>quotient and remainder</td>
</tr>
<tr>
<td></td>
<td>mod</td>
<td>modulo</td>
</tr>
<tr>
<td></td>
<td>rem</td>
<td>remainder; different sign matching rules</td>
</tr>
<tr>
<td>scale/shift</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ssu</td>
<td>static scale up</td>
</tr>
<tr>
<td></td>
<td>ssd</td>
<td>static scale down with rounding</td>
</tr>
<tr>
<td></td>
<td>dsu</td>
<td>dynamic scale up</td>
</tr>
<tr>
<td></td>
<td>dsd</td>
<td>dynamic scale down with rounding</td>
</tr>
<tr>
<td></td>
<td>lsh</td>
<td>left shift</td>
</tr>
<tr>
<td></td>
<td>ash</td>
<td>arithmetic right shift</td>
</tr>
<tr>
<td></td>
<td>rsh</td>
<td>logical right shift</td>
</tr>
<tr>
<td></td>
<td>rtl</td>
<td>rotate left</td>
</tr>
<tr>
<td></td>
<td>rtr</td>
<td>rotate right</td>
</tr>
<tr>
<td>composite</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add2</td>
<td>add with two addends</td>
</tr>
<tr>
<td></td>
<td>sub2</td>
<td>subtract with two subtrahends</td>
</tr>
<tr>
<td></td>
<td>lshadd</td>
<td>left shift and add</td>
</tr>
<tr>
<td></td>
<td>lshsub</td>
<td>left shift and sub</td>
</tr>
<tr>
<td></td>
<td>muladd</td>
<td>multiply and add</td>
</tr>
<tr>
<td></td>
<td>mulsub</td>
<td>multiply and subtract</td>
</tr>
<tr>
<td></td>
<td>muladd2</td>
<td>multiply and add with two addends</td>
</tr>
<tr>
<td></td>
<td>mulsub2</td>
<td>multiply and subtract with two subtrahends</td>
</tr>
<tr>
<td>bitwise</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>not</td>
<td>1’s complement</td>
</tr>
<tr>
<td></td>
<td>and</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td>xor</td>
<td>exclusive or</td>
</tr>
<tr>
<td></td>
<td>ior</td>
<td>inclusive or</td>
</tr>
<tr>
<td></td>
<td>dif</td>
<td>difference (a&amp;~b)</td>
</tr>
</tbody>
</table>
3 Machine Abstraction Layer

The machine abstraction layer enables a mostly machine-independent implementation of this specification.

In what header the following functions are defined?

3.1 Overflow-Detecting Operations

The overflow-detecting functions return a boolean true when the operation overflows, and a boolean false when the operation does not overflow. Compilers may assume that a true result is rare. When the return is false, the function writes the operation result through the given pointer. When the return is true, the pointer is not used and no write occurs.

The following functions are available. Within these prototypes T and C are any integer type. However, C is useful only when it does not have values that T has.

For the above paragraph:

— T and C are too short names. They should be Integer, Integer1, Integer2.
— Should those functions be more generic and work with wide_integer and integer? If not, then we can use the concept Integral instead.

Functions below could be used as a basic building block for wide_integer, however they miss a few important things:

— constexpr - almost all the operations on wide_integer are constexpr. Without constexpr the below functions are not usable with wide_integer.
— noexcept - changing the pointer argument to a reference makes it impossible to pass a nullptr and to get an UB. This is quite helpful for users and makes the below functions suit better the wide_integer needs.

```c
constexpr bool overflow_neg(T*& result, T value) noexcept;
constexpr bool overflow_lsh(T*& product, T multiplicand, int count) noexcept;
constexpr bool overflow_add(T*& summand, T augend, T addend) noexcept;
constexpr bool overflow_sub(T*& difference, T minuend, T subtrahend) noexcept;
constexpr bool overflow_mul(T*& product, T multiplicand, T multiplier) noexcept;
```

3.2 Overflow-Handling Operations

Overflow-handling operations require an overflow handling mode. We represent the mode in C++ as an enumeration:

```c
enum class overflow {
    impossible, undefined, abort, exception,
    special,
    saturate, modulo_shifted
};
```
Within the definition of the following functions, we use a defining function, which we do not expect will be directly represented in C++. It is \( T \text{overflow}(\text{mode}, T \text{ lower}, T \text{ upper}, U \text{ value}) \) where \( U \) either

- has a range that is not a subset of the range of \( T \) or
- is evaluated as a real number expression.

Many C++ conversions already reduce the range of a value, but they do not provide programmer control of that reduction. We can give programmers control.

\[
\text{template<typename T, typename U> constexpr T convert(overflow mode, U value);} \\
\text{Returns: convert(mode, lower, upper, value).}
\]

Being able to specify overflow from a range of values of the same type is also helpful.

\[
\text{template<typename T> constexpr T limit(overflow mode, T lower, T upper, T value);} \\
\text{Returns: limit(mode, lower, upper, value).}
\]

Common arguments can be elided with convenience functions.

\[
\text{template<typename T> constexpr T limit_nonnegative(overflow mode, T upper, T value);} \\
\text{Returns: limit(mode, 0, upper, value).}
\]

\[
\text{template<typename T> constexpr T limit_signed(overflow mode, T upper, T value);} \\
\text{Returns: limit(mode, -upper, upper, value).}
\]

Two's-complement numbers are a slight variant on the above.

\[
\text{template<typename T> constexpr T limit_twoscomp(overflow mode, T upper, T value);} \\
\text{Returns: limit_twoscomp(mode, upper, value).}
\]

For binary representations, we can also specify bits instead. While this specification may seem redundant, it enables faster implementations.

\[
\text{template<typename T> constexpr T limit_nonnegative_bits(overflow mode, T upper, T value);} \\
\text{Returns: overflow(mode, 0, 2^{upper} - 1, value).}
\]
template<overflow mMode, typename T> constexpr T limit_nonnegative_bits(T upper, T value);

Returns: limit_nonnegative_bits(mMode, upper, value).

template<typename T> constexpr T limit_signed_bits(overflow mode, T upper, T value);

Returns: overflow(mode, -\(2^{upper} - 1\), \(2^{upper} - 1\), value).

template<overflow mMode, typename T> constexpr T limit_signed_bits(T upper, T value);

Returns: limit_signed_bits(mMode, upper, value).

template<typename T> constexpr T limit_twoscomp_bits(overflow mode, T upper, T value);

Returns: overflow(mode, -\(2^{upper}\), \(2^{upper} - 1\), value).

template<overflow mMode, typename T> constexpr T limit_twoscomp_bits(T upper, T value);

Returns: limit_twoscomp_bits(mMode, upper, value).

Embedding overflow detection within regular operations can lead to enhanced performance. In particular, left shift is a important candidate operation within fixed-point arithmetic.

template<typename T> constexpr T scale_up(overflow mode, T value, int count);

Returns: overflow(mode, numeric_limits<T>::min(), numeric_limits<T>::max(), value*\(2^{count}\)).

template<overflow mMode, typename T> constexpr T scale_up(T value, int count);

Returns: scale_up(mMode, value, count).

3.3 Rounding Operations

We represent the rounding mode in C++ as an enumeration:

```cpp
enum class rounding {
    all_to_neg_inf, all_to_pos_inf,
    all_to_zero, all_away_zero,
    all_to_even, all_to_odd,
    all_fastest, all_smallest,
    all_unspecified,
    tie_to_neg_inf, tie_to_pos_inf,
    tie_to_zero, tie_away_zero,
    tie_to_even, tie_to_odd,
    tie_fastest, tie_smallest,
    tie_unspecified
};
```

The unmotivated modes all_away_zero, all_to_even, all_to_odd, tie_to_neg_inf, tie_to_pos_inf, and tie_to_zero are conditionally supported.

Within the definition of the following functions, we use a defining function, which we do not expect will be directly represented in C++. It is T round(mode, U) where U either

- has a finer resolution than T or
- is evaluated as a real number expression.

We already have rounding functions for converting floating-point numbers to integers. However, we need a facility that extends to different sizes of floating-point and between other numeric types.

template<typename T, typename U> T convert(rounding mode, U value);
3.4 Combined Rounding and Overflow Operations

Some operations may reasonably both require rounding and require overflow detection. First and foremost, conversion from floating-point to integer may require handling a floating-point value that has both a finer resolution and a larger range than the integer can handle. The problem generalizes to arbitrary numeric types.

3.5 Double-Word Operations

There are two classes of functions, those that provide a result in a single double-wide type and those that provide a result split into two single-wide types. We expect programmers to use type names from `<cstdint>` or the parametric type aliases (below). Hence, we do not need to provide a means to infer one type size from the other. Within this section, we name these types as follows.
Table 2 — Double-word operations

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>signed integer type</td>
</tr>
<tr>
<td>U</td>
<td>unsigned integer type</td>
</tr>
<tr>
<td>DS</td>
<td>signed integer type that is double the width of the S type</td>
</tr>
<tr>
<td>DU</td>
<td>unsigned integer type that is double the width of the U type</td>
</tr>
</tbody>
</table>

We need a mechanism to specify the largest supported type for various combinations of function category and operation category. To that end, we propose macros as follows.

Macros will stop working with modules. Instead of a macro we have to use aliases.

What names should we have for them and do we need to put them into a separate namespace?

Table 3 — Macros

<table>
<thead>
<tr>
<th>macro name</th>
<th>result category</th>
<th>operation category</th>
</tr>
</thead>
<tbody>
<tr>
<td>LARGEST_DOUBLE__WIDE_ADD</td>
<td>double-wide</td>
<td>add, add2, sub, sub2</td>
</tr>
<tr>
<td>LARGEST_DOUBLE__WIDE_LSH</td>
<td>double-wide</td>
<td>lsh, lshadd</td>
</tr>
<tr>
<td>LARGEST_DOUBLE__WIDE_MUL</td>
<td>double-wide</td>
<td>mul, muladd, muladd2, mulsub, mulsub2</td>
</tr>
<tr>
<td>LARGEST_DOUBLE__WIDE_DIV</td>
<td>double-wide</td>
<td>divn, divw, divnrem, divwrem</td>
</tr>
<tr>
<td>LARGEST_DOUBLE__WIDE_ALL</td>
<td>double-wide</td>
<td>the minimum size of the four macros above</td>
</tr>
<tr>
<td>LARGEST_SINGLE__WIDE_ADD</td>
<td>single-wide</td>
<td>add, add2, sub, sub2</td>
</tr>
<tr>
<td>LARGEST_SINGLE__WIDE_LSH</td>
<td>single-wide</td>
<td>lsh, lshadd</td>
</tr>
<tr>
<td>LARGEST_SINGLE__WIDE_MUL</td>
<td>single-wide</td>
<td>mul, muladd, muladd2, mulsub, mulsub2</td>
</tr>
<tr>
<td>LARGEST_SINGLE__WIDE_DIV</td>
<td>single-wide</td>
<td>divn, divw, divnrem, divwrem</td>
</tr>
<tr>
<td>LARGEST_SINGLE__WIDE_ALL</td>
<td>double-wide</td>
<td>the minimum size of the four macros above</td>
</tr>
</tbody>
</table>

We need a mechanism to build and split double-wide types. The lower part of the split is always an unsigned type.

```cpp
constexpr S split_upper(DS value) noexcept;
constexpr U split_lower(DS value) noexcept;
constexpr DS wide_build(S upper, U lower) noexcept;
```

§ 3.5
constexpr U split_upper(DU value) noexcept;
constexpr U split_lower(DU value) noexcept;
constexpr DU wide_build(U upper, U lower) noexcept;

The arithmetic functions with an double-wide result are as follows. This category seems less important than the next category.

constexpr DS wide_lsh(S multiplicand, int count);
constexpr DS wide_add(S augend, S addend);
constexpr DS wide_sub(S minuend, S subtrahend);
constexpr DS wide_mul(S multiplicand, S multiplier);
constexpr DS wide_add2(S augend, S addendi, S addend2);
constexpr DS wide_sub2(S minuend, S subtrahend1, S subtrahend2);
constexpr DS wide_lshadd(S multiplicand, int count, S addend);
constexpr DS wide_muladd(S multiplicand, S multiplier, S addend);
constexpr DS wide_mulsub(S multiplicand, S multiplier, S subtrahend);
constexpr DS wide_muladd2(S multiplicand, S multiplier, S addendi, S addend2);
constexpr DS wide_mulsub2(S multiplicand, S multiplier, S subtrahend1, S subtrahend2);
constexpr S wide_divn(DS dividend, S divisor);
constexpr DS wide_divw(DS dividend, S divisor);
constexpr S wide_divnrem(S & remainder, DS dividend, S divisor);
constexpr DS wide_divnwrem(S & remainder, DS dividend, S divisor);

constexpr DU wide_lsh(U multiplicand, int count) noexcept;
constexpr DU wide_add(U augend, U addend) noexcept;
constexpr DU wide_sub(U minuend, U subtrahend) noexcept;
constexpr DU wide_mul(U multiplicand, U multiplier) noexcept;
constexpr DU wide_add2(U augend, U addendi, U addend2) noexcept;
constexpr DU wide_sub2(U minuend, U subtrahend1, U subtrahend2) noexcept;
constexpr DU wide_lshadd(U multiplicand, int count, U addend) noexcept;
constexpr DU wide_muladd(U multiplicand, U multiplier, U addend) noexcept;
constexpr DU wide_mulsub(U multiplicand, U multiplier, U subtrahend) noexcept;
constexpr DU wide_muladd2(U multiplicand, U multiplier, U addendi, U addend2) noexcept;
constexpr DU wide_mulsub2(U multiplicand, U multiplier, U subtrahend1, U subtrahend2) noexcept;
constexpr U wide_divn(DU dividend, U divisor);
constexpr DU wide_divw(DU dividend, U divisor);
constexpr U wide_divnrem(U & remainder, DU dividend, U divisor);
constexpr DU wide_divnwrem(U & remainder, DU dividend, U divisor);

The arithmetic functions with a split result are as follows. The lower part of the result is always an unsigned type. The lower part is returned through a pointer while the upper part is returned as the function result. The intent is that in loops, the lower part is written once to memory while the upper part is carried between iterations in a local variable.

Do the below functions have wide or narrow contract? Should they be noexcept?

constexpr S split_lsh(U & product, S multiplicand, int count);
constexpr S split_add(U & summand, S augend, S addend);
constexpr S split_sub(U & difference, S minuend, S subtrahend);
constexpr S split_mul(U & product, S multiplicand, S multiplier);
constexpr S split_add2(U & summand, S value1, S addendi, S addend2);
constexpr S split_sub2(U & difference, S minuend, S subtrahend1, S subtrahend2);
constexpr S split_lshadd(U & product, S multiplicand, int count, S addend);
constexpr S split_lahsub(U & product, S multiplicand, int count, S subtrahend);
constexpr S split_muladd(U& product, S multiplicand, S addend1, S addend);
constexpr S split_mulsub(U& product, S multiplicand, S subtrahend1, S subtrahend2);
constexpr S split_muladd2(U& product, S multiplicand, S multiplier, S addend1, S addend2);
constexpr S split_mulsub2(U& product, S multiplicand, S multiplier, S subtrahend1, S subtrahend2);
constexpr S split_divn(S dividend_upper, U dividend_lower, S divisor);
constexpr DS split_divw(S dividend_upper, U dividend_lower, S divisor);
constexpr S split_divnrem(S& remainder, S dividend_upper, U dividend_lower, S divisor);
constexpr DS split_divwrem(S& remainder, S dividend_upper, U dividend_lower, S divisor);

constexpr U split_lsh(U& product, U multiplicand, int count);
constexpr U split_add(U& summand, U value1, U addend);
constexpr U split_sub(U& difference, U minuend, U subtrahend);
constexpr U split_mul(U& product, U multiplicand, U multiplier);
constexpr U split_add2(U& summand, U value1, U addend1, U addend2);
constexpr U split_sub2(U& difference, U minuend, U subtrahend1, U subtrahend2);
constexpr U split_lshadd(U& product, U multiplicand, int count, U addend);
constexpr U split_lshsub(U& product, U multiplicand, int count, U subtrahend);
constexpr U split_muladd(U& product, U multiplicand, U multiplier, U addend);
constexpr U split_mulsub(U& product, U multiplicand, U multiplier, U subtrahend);
constexpr U split_muladd2(U& product, U multiplicand, U multiplier, U addend1, U addend2);
constexpr U split_mulsub2(U& product, U multiplicand, U multiplier, U subtrahend1, U subtrahend2);
constexpr U split_divn(U dividend_upper, U dividend_lower, U divisor);
constexpr DU split_divw(U dividend_upper, U dividend_lower, U divisor);
constexpr U split_divnrem(U& remainder, U dividend_upper, U dividend_lower, U divisor);
constexpr DU split_divwrem(U& remainder, U dividend_upper, U dividend_lower, U divisor);
4 Machine extension layer
[machine_ext_layer]

The machine extension layer enables the implementation of extended types.

4.1 Word-array operations [word_array_ops]
C++20 will have a std::span that is designed to be a replacement for pointer* + size. It provides additional advantages:

— bounds may be checked at compile time
— compiler could unroll the internal loops knowing the bounds at compile time (as the bounds are known in case of wide_integer)
— std::span is more simple to use with continuous containers and arrays
— std::span protects from nullptr

Should it be used here?

A word is the type provided by LARGEST_SINGLE_WIDE_ALL and defined above.

We provide the following operations. These operations are not intended to provide complete multi-word operations, but rather to handle subarrays with uniform operations. Higher-level operations then compose these operations into a complete operation.

wide_integer could benefit from overloads that have an array version of last parameter, like U* addend, int addend_length.

constexpr U unsigned_subarray_addin_word(U* multiplicand, int length, U addend);

    Expects: [multiplicand, multiplicand + length) is a valid range.
    Effects: Add the word addend to the multiplicand of length length, leaving the result in the multiplicand.
    Returns: Any carry out from the accumulator.

constexpr U unsigned_subarray_add_word(U* summand, const U* augend, int length, U addend);

    Expects: [multiplicand, multiplicand + length) is a valid range. [augend, augend + length) is a valid range.
    Effects: Add the addend to the augend of length length, writing the result to the summand, which is also of length length.
    Returns: Any carry out from the summand.

constexpr U unsigned_subarray_mulin_word(U* product, int length, U multiplier);

    Expects: [product, product + length) is a valid range.
    Effects: Multiply the product of length length by the multiplier, leaving the result in the product.
    Returns: Any carry out from the product.
constexpr U unsigned_subarray_mul_word(U* product, const U* multiplicand, int length, U multiplier);

    Expects: [product, product + length) is a valid range. [multiplicand, multiplicand + length) is a valid range.
    Effects: Multiply the multiplicand of length length by the multiplier writing the result to the product, which is also of length length.
    Returns: Any carry out from the product.

constexpr U unsigned_subarray_accmul_word(U* accumulator, const U* multiplicand, int length, U multiplier);

    Expects: [accumulator, accumulator + length) is a valid range. [multiplicand, multiplicand + length) is a valid range.
    Effects: Multiply the multiplicand of length length by the multiplier adding the result to the accumulator, which is also of length length.
    Returns: Any carry out from the accumulator.

For each of the two add operations above, there is a corresponding subtract operation.
For each of the seven operations above (add+sub+mul), there is a corresponding signed operation. The primary difference between the two is sign extension.
For each of the fourteen operations in above, there is a corresponding operation where the 'right-hand' argument is a pointer to a subarray, which is also of length length.
Each big numeric addition should have a feature testing macro. Not sure about a feature testing macro for the whole Numbers TS.

If an implementation supplies all of the conditionally supported features specified in `<wide_integer>` header in this document shall additionally define the `__cpp_lib_wide_integer` feature test macro.

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__cpp_lib_wide_integer</code></td>
<td>201909</td>
</tr>
<tr>
<td><code>__cpp_lib_integer</code></td>
<td>201909</td>
</tr>
<tr>
<td><code>__cpp_lib_numericAliases</code></td>
<td>201909</td>
</tr>
<tr>
<td><code>__cpp_lib_numeric_machine_layer</code></td>
<td>201909</td>
</tr>
<tr>
<td><code>__cpp_lib_rational</code></td>
<td>201909</td>
</tr>
</tbody>
</table>
6 Wide Integers

6.1 Class template numeric_limits

Add the following sentence after the sentence "Specializations shall be provided for each arithmetic type, both floating-point and integer, including bool." (first sentence in fourth paragraph in numeric_limits):

Specializations shall be also provided for wide_integer type.

[Note: If there is a built-in integral type Integral that has the same signedness and width as wide_integer<Bits, S>, then numeric_limits<wide_integer<Bits, S>> specialized in the same way as numeric_limits<Integral> — end note]

This is the only paper that adjusts numeric_limits. Should the other papers adjust numeric_limits too?

6.2 Header <wide_integer> synopsis

namespace std {

   // 26.2.2 class template wide_integer
   template<size_t Bits, typename S> class wide_integer;

   // 26.2.2.1 type traits specializations
   template<size_t Bits, typename S, size_t Bits2, typename S2>
      struct common_type<wide_integer<Bits, S>, wide_integer<Bits2, S2>>;

   template<size_t Bits, typename S, typename Arithmetic>
      struct common_type<wide_integer<Bits, S>, Arithmetic>;

   template<typename Arithmetic, size_t Bits, typename S>
      struct common_type<Arithmetic, wide_integer<Bits, S>>:
         common_type<wide_integer<Bits, S>, Arithmetic>;

   // 26.2.2.2 unary operations
   template<size_t Bits, typename S>
      constexpr wide_integer<Bits, S> operator~(const wide_integer<Bits, S>& val) noexcept;

   template<size_t Bits, typename S>
      constexpr wide_integer<Bits, S> operator-(const wide_integer<Bits, S>& val) noexcept(is_unsigned_v<S>);

   template<size_t Bits, typename S>
      constexpr wide_integer<Bits, S> operator+(const wide_integer<Bits, S>& val) noexcept(is_unsigned_v<S>);

   // 26.2.2.3 binary operations
   template<size_t Bits, typename S, size_t Bits2, typename S2>
      common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
         operator*(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

   template<size_t Bits, typename S, size_t Bits2, typename S2>
      common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
         operator/(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

   template<size_t Bits, typename S, size_t Bits2, typename S2>
      common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
         operator/(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

   template<size_t Bits, typename S, size_t Bits2, typename S2>
      common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
         operator/(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

§ 6.2
constexpr operator+(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept(is_unsigned_v<S>);

template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
constexpr operator-(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
constexpr operator%(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
constexpr operator&(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
constexpr operator|(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

template<size_t Bits, typename S, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
constexpr operator^(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

template<size_t Bits, typename S>
common_type_t<wide_integer<Bits, S>, size_t>
constexpr operator<<(const wide_integer<Bits, S>& lhs, size_t rhs);

template<size_t Bits, typename S>
common_type_t<wide_integer<Bits, S>, size_t>
constexpr operator>>(const wide_integer<Bits, S>& lhs, size_t rhs);

The spaceship operator should be used here and in other papers.

template<size_t Bits, typename S, size_t Bits2, typename S2>
constexpr bool operator<(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

template<size_t Bits, typename S, size_t Bits2, typename S2>
constexpr bool operator>(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

template<size_t Bits, typename S, size_t Bits2, typename S2>
constexpr bool operator<=(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

template<size_t Bits, typename S, size_t Bits2, typename S2>
constexpr bool operator>=(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

template<size_t Bits, typename S, size_t Bits2, typename S2>
constexpr bool operator==(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

template<size_t Bits, typename S, size_t Bits2, typename S2>
constexpr bool operator!=(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

// 26.?? numeric conversions

template<size_t Bits, typename S> std::string to_string(const wide_integer<Bits, S>& val);

template<size_t Bits, typename S> std::wstring to_wstring(const wide_integer<Bits, S>& val);

§ 6.2
The header `<wide_integer>` defines class template `wide_integer` and a set of operators for representing and manipulating integers of specified width.

Example:

```cpp
using int128_t = wide_int<128>;
constexpr int128_t c = std::numeric_limits<int128_t>::min();
static_assert(c == 0x80000000000000000000000000000000_uint128);

int256_t a = 13;
a += 0xFF;
```
6.3 Template class wide_integer overview

namespace std {
    template<size_t Bits, typename S>
    class wide_integer {
    public:
        // 26.???.2.?? construct:
        constexpr wide_integer() noexcept = default;
        constexpr wide_integer(const wide_integer<Bits, S>& ) noexcept = default;
        template<typename Arithmetic> constexpr wide_integer(const Arithmetic& other) noexcept;
        template<size_t Bits2, typename S2> constexpr wide_integer(const wide_integer<Bits2, S2>& other) noexcept;

        // 26.???.2.?? assignment:
        constexpr wide_integer<Bits, S>& operator=(const wide_integer<Bits, S>& ) noexcept = default;
        template<typename Arithmetic>
            constexpr wide_integer<Bits, S>& operator=(const Arithmetic& other) noexcept;
        template<size_t Bits2, typename S2>
            constexpr wide_integer<Bits, S>& operator=(const wide_integer<Bits2, S2>& other) noexcept;

        // 26.???.2.?? compound assignment:
        template<typename Arithmetic>
            constexpr wide_integer<Bits, S>& operator*=(const Arithmetic&);
        template<size_t Bits2, typename S2>
            constexpr wide_integer<Bits, S>& operator*=(const wide_integer<Bits2, S2>&);

        template<typename Arithmetic>
            constexpr wide_integer<Bits, S>& operator/=(const Arithmetic&);
        template<size_t Bits2, typename S2>
            constexpr wide_integer<Bits, S>& operator/=(const wide_integer<Bits2, S2>&);

        template<typename Integral>
            constexpr wide_integer<Bits, S>& operator%=(const Integral&);
        template<size_t Bits2, typename S2>
            constexpr wide_integer<Bits, S>& operator%=(const wide_integer<Bits2, S2>&);

        template<typename Integral>
            constexpr wide_integer<Bits, S>& operator&=(const Integral&)
            noexcept(is_unsigned_v<S>);
        template<size_t Bits2, typename S2>
            constexpr wide_integer<Bits, S>& operator&=(const wide_integer<Bits2, S2>&)
            noexcept(is_unsigned_v<S>);

        template<typename Integral>
            constexpr wide_integer<Bits, S>& operator|=(const Integral&)
            noexcept;
        template<size_t Bits2, typename S2>
            constexpr wide_integer<Bits, S>& operator|=(const wide_integer<Bits2, S2>&);

        constexpr wide_integer<Bits, S>& operator|=(const Integral&)
            noexcept(is_unsigned_v<S>);

        constexpr wide_integer<Bits, S>& operator|=(const wide_integer<Bits2, S2>&)
            noexcept(is_unsigned_v<S>);

        constexpr wide_integer<Bits, S>& operator|=(const wide_integer<Bits2, S2>&)
            noexcept(is_unsigned_v<S>);

        constexpr wide_integer<Bits, S>& operator|=(const Integral&)
            noexcept;

        constexpr wide_integer<Bits, S>& operator|=(const wide_integer<Bits2, S2>&)
            noexcept;

        constexpr wide_integer<Bits, S>& operator|=(const wide_integer<Bits2, S2>&)
            noexcept(is_unsigned_v<S>);
    }
}

§ 6.3
The class template `wide_integer<`size_t` Bits, typename S>` is a trivial standard layout class that behaves as an integer type of a compile time specified bitness.

Template parameter `Bits` specifies exact bits count to store the integer value. `Bits`

When size of `wide_integer` is equal to a size of builtin integral type then the alignment and layout of that `wide_integer` is equal to the alignment and layout of the builtin type.

Template parameter `S` specifies signedness of the stored integer value and is either signed or unsigned.

Implementations are permitted to add explicit conversion operators and explicit or implicit constructors for `Arithmetic` and for `Integral` types.

Example:

```cpp
template <class Arithmetic>
[
[[deprecated("Implicit conversions to builtin arithmetic types are not safe!")]]
    constexpr operator Arithmetic() const noexcept;
```

```
explicit constexpr operator bool() const noexcept;
```

```
explicit constexpr operator int() const noexcept;
```

...
template<typename Arithmetic>
constexpr wide_integer(const Arithmetic& other) noexcept;

Effects: Constructs an object from other using the integral conversion rules [conv.integral].

template<size_t Bits2, typename S2>
constexpr wide_integer(const wide_integer<Bits2, S2>& other) noexcept;

Effects: Constructs an object from other using the integral conversion rules [conv.integral].

6.3.2 wide_integer assignments [numeric.wide_integer.assign]

template<typename Arithmetic>
constexpr wide_integer<Bits, S>& operator=(const Arithmetic& other) noexcept;

Effects: Constructs an object from other using the integral conversion rules [conv.integral].

template<size_t Bits2, typename S2>
constexpr wide_integer<Bits, S>& operator=(const wide_integer<Bits2, S2>& other) noexcept;

Effects: Constructs an object from other using the integral conversion rules [conv.integral].

6.3.3 wide_integer compound assignments [numeric.wide_integer.cassign]

template<size_t Bits2, typename S2>
constexpr wide_integer<Bits, S>& operator*=(const wide_integer<Bits2, S2>&);

template<size_t Bits2, typename S2>
constexpr wide_integer<Bits, S>& operator/=(const wide_integer<Bits2, S2>&);

template<size_t Bits2, typename S2>
constexpr wide_integer<Bits, S>& operator+=(const wide_integer<Bits2, S2>&) noexcept(is_unsigned_v<S>);

template<size_t Bits2, typename S2>
constexpr wide_integer<Bits, S>& operator-=(const wide_integer<Bits2, S2>&) noexcept(is_unsigned_v<S>);

template<size_t Bits2, typename S2>
constexpr wide_integer<Bits, S>& operator%=(const wide_integer<Bits2, S2>&);

template<size_t Bits2, typename S2>
constexpr wide_integer<Bits, S>& operator&=(const wide_integer<Bits2, S2>&) noexcept;

template<size_t Bits2, typename S2>
constexpr wide_integer<Bits, S>& operator|=(const wide_integer<Bits2, S2>&) noexcept;

template<size_t Bits2, typename S2>
constexpr wide_integer<Bits, S>& operator^=(const wide_integer<Bits2, S2>&) noexcept;

template<size_t Bits2, typename S2>
constexpr wide_integer<Bits, S>& operator<<=(const wide_integer<Bits2, S2>&);

template<size_t Bits2, typename S2>
constexpr wide_integer<Bits, S>& operator>>=(const wide_integer<Bits2, S2>&) noexcept;

Effects: Behavior of the above operators is similar to operators for built-in integral types.

template<typename Arithmetic>
constexpr wide_integer<Bits, S>& operator*=(const Arithmetic&);

template<typename Arithmetic>
constexpr wide_integer<Bits, S>& operator/=(const Arithmetic&);

template<typename Arithmetic>
constexpr wide_integer<Bits, S>& operator+=(const Arithmetic&) noexcept(is_unsigned_v<S>);

template<typename Arithmetic>
constexpr wide_integer<Bits, S>& operator-=(const Arithmetic&) noexcept(is_unsigned_v<S>);

§ 6.3.3
Effects: As if an object wi of type `wide_integer<Bits, S>` was created from input value and the corresponding operator was called for `*this` and the wi.

```cpp
template<typename Integral>
constexpr wide_integer<Bits, S>& operator%=(const Integral&);

template<typename Integral>
constexpr wide_integer<Bits, S>& operator&=(const Integral&) noexcept;

template<typename Integral>
constexpr wide_integer<Bits, S>& operator|=(const Integral&) noexcept;

template<typename Integral>
constexpr wide_integer<Bits, S>& operator^=(const Integral&) noexcept;

template<typename Integral>
constexpr wide_integer<Bits, S>& operator<<=(const Integral&);

template<typename Integral>
constexpr wide_integer<Bits, S>& operator>>=(const Integral&) noexcept;
```

Effects: As if an object wi of type `wide_integer<Bits, S>` was created from input value and the corresponding operator was called for `*this` and the wi.

### 6.3.4 wide_integer observers

```cpp
template <typename Arithmetic> constexpr operator Arithmetic() const noexcept;
```

*Returns:* If `is_integral_v<Arithmetic>` then `Arithmetic` is constructed from `*this` using the integral conversion rules [conv.integral]. If `is_floating_point_v<Arithmetic>`, then `Arithmetic` is constructed from `*this` using the floating-integral conversion rules [conv.fpint]. Otherwise the operator shall not participate in overload resolution.

### 6.4 Specializations of common_type

```cpp
template<size_t Bits, typename S, size_t Bits2, typename S2>
struct common_type<wide_integer<Bits, S>, wide_integer<Bits2, S2>> {
  using type = wide_integer<max(Bits, Bits2), see below>;
};
```

The signed template parameter indicated by this specialization is following:

- `(is_signed_v<S> && is_signed_v<S2> ? signed : unsigned) if Bits == Bits2`
- `S` if `Bits > Bits2`
- `S2` otherwise

[Note: common_type follows the usual arithmetic conversions design. - end note]

[Note: common_type attempts to follow the usual arithmetic conversions design here for interoperability between different numeric types. Following two specializations must be moved to a more generic place and enriched with usual arithmetic conversion rules for all the other numeric classes that specialize `std::numeric_limits` - end note]

```cpp
template<size_t Bits, typename S, typename Arithmetic>
struct common_type<wide_integer<Bits, S>, Arithmetic> {
  using type = see below;
};
```

```cpp
template<typename Arithmetic, size_t Bits, typename S>
struct common_type<Arithmetic, wide_integer<Bits, S>> 
  : common_type<wide_integer<Bits, S>, Arithmetic>;
```
The member typedef type is following:

- Arithmetic if `numeric_limits<Arithmetic>::is_integer` is false
- `wide_integer<Bits, S>` if `sizeof(wide_integer<Bits, S>) > sizeof(Arithmetic)`
- Arithmetic if `sizeof(wide_integer<Bits, S>) < sizeof(Arithmetic)`
- Arithmetic if `sizeof(wide_integer<Bits, S>) == sizeof(Arithmetic) && is_signed_v<S>`
- Arithmetic if `sizeof(wide_integer<Bits, S>) == sizeof(Arithmetic) && numeric_limits<wide_integer<Bits, S>>::is_signed == numeric_limits<Arithmetic>::is_signed`
- `wide_integer<Bits, S>` otherwise

### 6.5 Unary operators [numeric.wide_integer.unary_ops]

```c++
#include <numeric>

template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
operator*(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

Returns: CT(lhs) *= rhs.
```

```c++
#include <numeric>

template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
operator/(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

Returns: CT(lhs) /= rhs.
```

```c++
#include <numeric>

template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
operator+(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

Returns: CT(lhs) += rhs.
```

```c++
#include <numeric>

template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
operator-(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

Returns: CT(lhs) -= rhs.
```

### 6.6 Binary operators [numeric.wide_integer.binary_ops]

In the function descriptions that follow, CT represents `common_type_t<A, B>`, where A and B are the types of the two arguments to the function.

```c++
#include <numeric>

template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
operator+(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

Returns: CT(lhs) += rhs.
```

```c++
#include <numeric>

template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
operator-(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

Returns: CT(lhs) -= rhs.
```

```c++
#include <numeric>

template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
operator*(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

Returns: CT(lhs) *= rhs.
```

```c++
#include <numeric>

template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
operator/(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

Returns: CT(lhs) /= rhs.
```

```c++
#include <numeric>

template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t<wide_integer<Bits, S>, wide_integer<Bits2, S2>>
operator+(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);

Returns: CT(lhs) += rhs.
```
\textit{Returns:} CT(lhs) \texttt{\textasciitilde}= rhs.

\begin{verbatim}
template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t\%(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);
\end{verbatim}
\textit{Returns:} CT(lhs) \texttt{\textasciitilde}= rhs.

\begin{verbatim}
template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t\&(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;
\end{verbatim}
\textit{Returns:} CT(lhs) \texttt{&}= rhs.

\begin{verbatim}
template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t\%(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);
\end{verbatim}
\textit{Returns:} CT(lhs) \texttt{\textasciitilde}= rhs.

\begin{verbatim}
template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t\%(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);
\end{verbatim}
\textit{Returns:} CT(lhs) \texttt{\textasciitilde}= rhs.

\begin{verbatim}
template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t\%(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);
\end{verbatim}
\textit{Returns:} CT(lhs) \texttt{\textasciitilde}= rhs.

\begin{verbatim}
template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t\%(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);
\end{verbatim}
\textit{Returns:} CT(lhs) \texttt{\textasciitilde}= rhs.

\begin{verbatim}
template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t\%(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);
\end{verbatim}
\textit{Returns:} CT(lhs) \texttt{\textasciitilde}= rhs.

\begin{verbatim}
template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t\%(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);
\end{verbatim}
\textit{Returns:} CT(lhs) \texttt{\textasciitilde}= rhs.

\begin{verbatim}
template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t\%(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);
\end{verbatim}
\textit{Returns:} CT(lhs) \texttt{\textasciitilde}= rhs.

\begin{verbatim}
template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t\%(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);
\end{verbatim}
\textit{Returns:} CT(lhs) \texttt{\textasciitilde}= rhs.

\begin{verbatim}
template<typename S, size_t Bits, size_t Bits2, typename S2>
common_type_t\%(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs);
\end{verbatim}
\textit{Returns:} CT(lhs) \texttt{\textasciitilde}= rhs.
Returns: true if value of CT(lhs) is equal or greater than the value of CT(rhs).

```
template<
  size_t Bits, typename S, size_t Bits2, typename S2>
constexpr bool operator==(const wide_integer<Bits, S>& lhs,
  const wide_integer<Bits2, S2>& rhs) noexcept;
```

Returns: true if significant bits of CT(lhs) and CT(rhs) are the same.

```
template<
  size_t Bits, typename S, size_t Bits2, typename S2>
constexpr bool operator!=(const wide_integer<Bits, S>& lhs,
  const wide_integer<Bits2, S2>& rhs) noexcept;
```

```
template<
  size_t Bits, typename S> std::string to_string(const wide_integer<Bits, S>& val);
```

```
template<
  size_t Bits, typename S> std::wstring to_wstring(const wide_integer<Bits, S>& val);
```

Returns: Each function returns an object holding the character representation of the value of its argument. All the significant bits of the argument are outputed as a signed decimal in the style [-]dddd.

```
template <
  size_t Bits, typename S>
  to_chars_result to_chars(char* first, char* last, const wide_integer<Bits, S>& value,
    int base = 10);
```

Behavior of wide_integer overload is subject to the usual rules of primitive numeric output conversion functions [utility.to.chars].

```
template <
  size_t Bits, typename S>
  from_chars_result from_chars(const char* first, const char* last, wide_integer<Bits, S>& value,
    int base = 10);
```

Behavior of wide_integer overload is subject to the usual rules of primitive numeric input conversion functions [utility.from.chars].

6.8 iostream specializations

```
template<class Char, class Traits, size_t Bits, typename S>
  basic_ostream<Char, Traits>& operator<<(basic_ostream<Char, Traits>& os,
    const wide_integer<Bits, S>& val);
```

1 Effects: As if by: os << to_string(val).

2 Returns: os.

```
template<class Char, class Traits, size_t Bits, typename S>
  basic_istream<Char, Traits>& operator>>(basic_istream<Char, Traits>& is,
    wide_integer<Bits, S>& val);
```

3 Effects: Extracts a wide_integer that is represented as a decimal number in the is. If bad input is encountered, calls is.setstate(ios_base::failbit) (which may throw ios::failure ([iostate.flags])).

4 Returns: is.

6.9 Hash support

```
template<
  size_t Bits, typename S> struct hash<wide_integer<Bits, S>>;
```

The specialization is enabled (20.14.14). If there is a built-in integral type Integral that has the same typename and width as wide_integer<Bits, S>, and wi is an object of type wide_integer<Bits, S>, then hash<wide_integer<MachineWords, S>>()(wi) == hash<Integral>()(Integral(wi)).
7 Rational math

7.1 Class rational

Default constructor of the class should be marked with `constexpr` so that the compiler could statically initialize it.

```cpp
class rational {
public:
    constexpr rational() noexcept;
    rational(const rational& rat);
    rational(rational&& rat) noexcept;
    explicit rational(float num);
    explicit rational(double num);
    explicit rational(long double num);
    explicit rational(integer num);
    rational(integer num, integer den);
    ~rational() noexcept;
    rational& operator=(const rational& rat);
    rational& operator=(rational&& rat) noexcept;
    rational& operator=(integer num);
    rational& assign(integer num, integer den);
    void swap(rational& rhs) noexcept;
    rational normalize() const;
    integer numer() const;
    integer denom() const;
    explicit operator bool() const noexcept;
    rational& negate() noexcept;
    rational& invert() noexcept;
    rational& operator++();
    rational& operator--();
    rational operator++(int);
    rational operator--(int);
    rational& operator+=(const integer& rhs);
    rational& operator-=(const integer& rhs);
    rational& operator*=(const integer& rhs);
    rational& operator/=(const integer& rhs);
}
```
rational& operator+=(const rational& rhs);
rational& operator-=(const rational& rhs);
rational& operator*=(const rational& rhs);
rational& operator/=(const rational& rhs);
};

The numerator and denominator shall be stored internally in a std::experimental::seminumeric::integer.

Probably to_chars, from_chars, I/O and std::hash overloads and specializations should be added.

rational member functions:
rational() noexcept;

Effects: Constructs a rational with a numerator equal to zero and a denominator equal to one.

rational(const rational& rat);
rational(rational&& rat);

Effects: Constructs a rational with a value of rat.

explicit rational(integer num);

Effects: Constructs a rational with a value of num.

explicit rational(float val);
explicit rational(double val);
explicit rational(long double val);

Effects: Constructs a rational with a value equal to val.

rational(integer num, integer den);

Requires: den != 0  Effects: Constructs a rational given the specified numerator and denominator.

~rational() noexcept;

Effects: Destructs *this.

rational& operator=(const rational& rhs);
rational& operator=(rational&& rhs) noexcept;

Effects: Assigns rhs to *this.
Returns: *this.

rational& operator=(integer num);

Effects: Assigns num to *this.
Returns: *this.

rational& assign(integer num, integer den);

Requires: den != 0
Effects: Assigns the specified numerator and denominator to *this.
Returns: *this.

void swap(rational& rhs) noexcept;

Effects: Swaps *this and rhs.
rational normalize() const;

    *Returns*: A rational equal to *this, but with the numerator and denominator having no common
factor other than 1 and the denominator greater than 0. If the numerator is 0, the denominator shall
be 1.

integer numer() const;

    *Returns*: The (possibly not normalized) numerator by value.

integer denom() const;

    *Returns*: The (possibly not normalized) denominator by value.

explicit operator bool() const noexcept;

    *Returns*: As if *this != 0.

rational& negate() noexcept;

    *Effects*: Changes the sign of *this.

    *Returns*: *this.

rational& invert() noexcept;

    *Requires*: the numerator is non-zero.

    *Effects*: Swaps the numerator and denominator.

    *Returns*: *this.

7.1.1 rational member operators:

rational& operator++();

    *Effects*: Adds 1 to *this and stores the result in *this.

    *Returns*: *this.

rational& operator--();

    *Effects*: Subtracts 1 from *this and stores the result in *this.

    *Returns*: *this.

rational operator++(int);

    *Effects*: Adds 1 to *this and stores the result in *this. *Returns*: The value of *this before the
addition.

rational operator--(int);

    *Effects*: Subtracts 1 from *this and stores the result in *this.

    *Returns*: The value of *this before the subtraction.

rational& operator+=(const integer& rhs);

    *Effects*: Adds the integer value rhs to *this and stores the result in *this.

    *Returns*: *this.

rational& operator-=(const integer& rhs);

§ 7.1.1
Effects: Subtracts the integer value \texttt{rhs} from \texttt{*this} and stores the result in \texttt{*this}.

Returns: \texttt{*this}.

\texttt{rational\& operator\*= (const integer\& rhs);} \\
Effects: Multiplies \texttt{*this} by the integer value \texttt{rhs} and stores the result in \texttt{*this}.

Returns: \texttt{*this}.

\texttt{rational\& operator/\=(const integer\& rhs);} \\
Requires: \texttt{rhs} \neq 0.

Effects: Divides \texttt{*this} by the integer value \texttt{rhs} and stores the result in \texttt{*this}.

Returns: \texttt{*this}.

\texttt{rational\& operator\+= (const rational\& rhs);} \\
Effects: Adds the rational value \texttt{rhs} to \texttt{*this} and stores the result in \texttt{*this}.

Returns: \texttt{*this}.

\texttt{rational\& operator\-= (const rational\& rhs);} \\
Effects: Subtracts the rational value \texttt{rhs} from \texttt{*this} and stores the result in \texttt{*this}.

Returns: \texttt{*this}.

\texttt{rational\& operator\*= (const rational\& rhs);} \\
Effects: Multiplies \texttt{*this} by the rational value \texttt{rhs} and stores the result in \texttt{*this}.

Returns: \texttt{*this}.

\texttt{rational\& operator/\=(const rational\& rhs);} \\
Requires: \texttt{rhs} \neq 0.

Effects: Divides \texttt{*this} by the rational value \texttt{rhs} and stores the result in \texttt{*this}.

Returns: \texttt{*this}.

\textbf{7.1.2 rational non-member operators:} \quad \text{[rational.ops]}

\texttt{rational\ operator\+(const rational\& val);} \\
Returns: \texttt{rational(val)}.

\texttt{rational\ operator\-(const rational\& val);} \\
Returns: \texttt{rational(val).negate()}.

\texttt{rational\ operator\+(const rational\& lhs, const rational\& rhs);} \\
Returns: \texttt{rational(lhs) \*= rhs}.

\texttt{rational\ operator\-(const rational\& lhs, const rational\& rhs);} \\
Returns: \texttt{rational(lhs) \-= rhs}.

\texttt{rational\ operator\*(const rational\& lhs, const rational\& rhs);} \\
Returns: \texttt{rational(lhs) \*= rhs}. 

\section*{§ 7.1.2}
rational operator/(const rational& lhs, const rational& rhs);
    
    Requires: rhs != 0.
    Returns: rational(lhs) /= rhs.

rational operator+(const rational& lhs, const integer& rhs);

Returns: rational(lhs) += rhs.

rational operator-(const rational& lhs, const integer& rhs);

Returns: rational(lhs) -= rhs.

rational operator*(const rational& lhs, const integer& rhs);

Returns: rational(lhs) *= rhs.

rational operator/(const rational& lhs, const integer& rhs);

Requires: rhs != 0.
Returns: rational(lhs) /= rhs.

rational operator+(const integer& lhs, const rational& rhs);

Returns: rational(rhs) += lhs.

rational operator-(const integer& lhs, const rational& rhs);


rational operator*(const integer& lhs, const rational& rhs);

Returns: rational(rhs) *= lhs.

rational operator/(const integer& lhs, const rational& rhs);

Requires: rhs != 0.
Returns: rational(rhs).invert() *= lhs.

bool operator==(const rational& lhs, const rational& rhs);

Returns: As if lhs.numer() * rhs.denom() == rhs.numer() * lhs.denom().

bool operator!=(const rational& lhs, const rational& rhs);

Returns: !(lhs == rhs).

bool operator<(const rational& lhs, const rational& rhs);

Returns: As if lhs.numer() * rhs.denom() < rhs.numer() * lhs.denom().

The spaceship operator should be used here and in other papers.

bool operator>(const rational& lhs, const rational& rhs);

Returns: rhs < lhs.

bool operator<=(const rational& lhs, const rational& rhs);

Returns: lhs < rhs || lhs == rhs.
bool operator>=(const rational& lhs, const rational& rhs);

*Returns*: \( \text{lhs} > \text{rhs} \mid\mid \text{lhs} == \text{rhs} \).

bool operator==(const rational& lhs, const integer& rhs);

*Returns*: As if after normalization \( \text{lhs.numer()} == \text{rhs} \) \&\& \( \text{lhs.denom()} == 1 \).

bool operator!=(const rational& lhs, const integer& rhs);

*Returns*: \(! (\text{lhs} == \text{rhs})\).

bool operator<(const rational& lhs, const integer& rhs);

*Returns*: As if \( \text{lhs.numer()} < \text{rhs} \ast \text{lhs.denom()} \).

bool operator>(const rational& lhs, const integer& rhs);

*Returns*: As if \( \text{lhs.numer()} > \text{rhs} \ast \text{lhs.denom()} \).

bool operator<=(const rational& lhs, const integer& rhs);

*Returns*: \( \text{lhs} < \text{rhs} \mid\mid \text{lhs} == \text{rhs} \).

bool operator>=(const rational& lhs, const integer& rhs);

*Returns*: \( \text{lhs} > \text{rhs} \mid\mid \text{lhs} == \text{rhs} \).

bool operator==(const integer& lhs, const rational& rhs);

*Returns*: \( \text{rhs} == \text{lhs} \).

bool operator!=(const integer& lhs, const rational& rhs);

*Returns*: \( \text{rhs} != \text{lhs} \).

bool operator<(const integer& lhs, const rational& rhs);

*Returns*: \( \text{rhs} > \text{lhs} \).

bool operator>(const integer& lhs, const rational& rhs);

*Returns*: \( \text{rhs} < \text{lhs} \).

bool operator<=(const integer& lhs, const rational& rhs);

*Returns*: \( \text{rhs} >= \text{lhs} \).

bool operator>=(const integer& lhs, const rational& rhs);

*Returns*: \( \text{rhs} <= \text{lhs} \).
8 Parametric aliases [parametric_aliases]

Parametric aliases provide a machine-independent mechanism to specify the desired allocation size built-in or extended types.

The following wording changes are relative to N4527.

8.1 Freestanding implementations [compliance]

Add the following entry to table 16.

<table>
<thead>
<tr>
<th>chapter</th>
<th>description</th>
<th>header</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.4+</td>
<td>Floating-point types</td>
<td>&lt;cstdfloat&gt;</td>
</tr>
</tbody>
</table>

8.2 General [support.general]

Add the following entry to table 29.

<table>
<thead>
<tr>
<th>chapter</th>
<th>description</th>
<th>header</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.4+</td>
<td>Floating-point types</td>
<td>&lt;cstdfloat&gt;</td>
</tr>
</tbody>
</table>

8.3 Header <cstdint> [cstdint.syn]

[Example:

```cpp
exact_2uint<128> distance(exact_2uint<64> a, exact_2uint<64> b);
exact_2uint<128> native_to_big_endian(exact_2uint<128> v);
```

First function does not relay on underlying implementation. Users would like to get the result in a `wide_integer` type if there’s no built-in 128bit type rather than getting an ill-formed program.

Second function does relay on bit representation of a number. Users would like to get an ill-formed program if the `exact_2uint<128>` does not has the same representation as a native type.

The question is: should we describe the layout of `wide_integer` type and make `*_2[u]int` return it if there’s no built-in type? Or we should leave the `*_2[u]int` as-is and do not deal with code portability for both cases? — end example]

Add the following entries to the synopsis before paragraph 1.

`MAX_BITS_2INT` and `MAX_BITS_2UINT` should be `inline constexpr int` variables. What name should they have?

```cpp
namespace std {
    template<int bits> alias using exact_2int = implementation-defined;
    template<int bits> alias using fast_2int = implementation-defined;
}
The aliases below are conditionally supported. The macro `MAX_BITS_2INT` shall give the largest integer size (in bits) supported by the aliases. [Note: All variants support the same sizes. âĂŤend note] If these aliases are not supported, the value shall be 0. Any parameter to the alias shall be in the range 1 to `MAX_BITS_2INT`.

```
template<int bits> alias using exact_2int = implementation-defined;
template<int bits> alias using fast_2int = implementation-defined;
template<int bits> alias using least_2int = implementation-defined;
```

```
#define MAX_BITS_2INT implementation-defined;
#define MAX_BITS_2UINT implementation-defined;
```

### 8.4 Parametric types

Add a new section with the following paragraphs.

The aliases below are conditionally supported. The macro `MAX_BITS_2INT` shall give the largest integer size (in bits) supported by the aliases. [Note: All variants support the same sizes. âĂŤend note] If these aliases are not supported, the value shall be 0. Any parameter to the alias shall be in the range 1 to `MAX_BITS_2INT`.

```
template<int bits> alias using exact_2int
```

The alias `exact_2int` refers to a built-in signed binary integer type of exactly `bits` bits. If there are two types of the same size, it refers to the type that is closest to `int` in promotion order. The type must represent negative values with two’s-complement representation.

```
template<int bits> alias using fast_2int
```

The alias `fast_2int` refers to the fastest built-in signed binary integer type of at least `bits` bits. If there are two types of the same size, it refers to the type that is closest to `int` in promotion order. The type must represent negative values with two’s-complement representation.

```
template<int bits> alias using least_2int
```

The alias `least_2int` refers to the smallest built-in signed binary integer type of at least `bits` bits. If there are two types of the same size, it refers to the type that is closest to `int` in promotion order. The type must represent negative values with two’s-complement representation.

```
template<int bits> alias using exact_2uint
```

The alias `exact_2uint` refers to a built-in unsigned binary integer type of exactly `bits` bits. If there are two types of the same size, it refers to the type that is closest to `unsigned int` in promotion order. The type must represent negative values with two’s-complement representation.

```
template<int bits> alias using fast_2uint
```

The alias `fast_2uint` refers to the fastest built-in unsigned binary integer type of at least `bits` bits. If there are two types of the same size, it refers to the type that is closest to `unsigned int` in promotion order.

```
template<int bits> alias using least_2uint
```

The alias `least_2uint` refers to the smallest built-in unsigned binary integer type of at least `bits` bits. If there are two types of the same size, it refers to the type that is closest to `unsigned int` in promotion order.

### 8.5 Floating-point types

After section 18.4, add a new section. It has no direct contents.
8.6 Header `<cstdfloat>`

Add a new section.

MAX_BITS_2IEEEFLOAT and MAX_BITS_10IEEEFLOAT should be inline constexpr int variables. What name should they have?

```cpp
namespace std {
    template<int bits>
        alias using exact_2ieeefloat = implementation-defined;
    template<int bits>
        alias using fast_2ieeefloat = implementation-defined;
    template<int bits>
        alias using least_2ieeefloat = implementation-defined;
    template<int bits>
        alias using exact_10ieeefloat = implementation-defined;
    template<int bits>
        alias using fast_10ieeefloat = implementation-defined;
    template<int bits>
        alias using least_10ieeefloat = implementation-defined;
}
#define MAX_BITS_2IEEEFLOAT implementation-defined;
#define MAX_BITS_10IEEEFLOAT implementation-defined;
```

8.7 Parametric types

Add a new section with the following paragraphs.

The aliases below are conditionally supported. The macro MAX_BITS_2IEEEFLOAT shall give the largest binary floating-point size (in bits) supported by the aliases. [Note: All variants support the same sizes. ©End note] If none of these aliases are supported, the value shall be 0. The parameter to the alias shall be in the range 1 to MAX_BITS_2IEEEFLOAT.

```cpp
template<int bits>
    alias using exact_2ieeefloat

    The alias exact_2ieeefloat refers to a built-in binary floating-point type of exactly bits bits. If there are two types of the same size, it refers to the type that is closest to double in promotion order. The type must use IEEE representation.

template<int bits>
    alias using fast_2ieeefloat

    The alias fast_2ieeefloat refers to the fastest built-in binary floating-point type of at least bits bits. If there are two types of the same size, it refers to the type that is closest to double in promotion order. The type must use IEEE representation.

template<int bits>
    alias using least_2ieeefloat

    The alias least_2ieeefloat refers to the smallest built-in binary floating-point type of at least bits bits. If there are two types of the same size, it refers to the type that is closest to double in promotion order. The type must use IEEE representation.
```

The aliases below are conditionally supported. The macro MAX_BITS_10IEEEFLOAT shall give the largest decimal floating point size (in bits) supported by the aliases. [Note: All variants support the same sizes. ©End note] If none of these aliases are supported, the value shall be 0. The parameter to the alias shall be in the range 1 to MAX_BITS_10IEEEFLOAT.

```cpp
template<int bits>
    alias using exact_10ieeefloat

    The alias exact_10ieeefloat refers to a built-in decimal floating-point type of exactly bits bits. If there are two types of the same size, it refers to the type that is closest to double in promotion order. The type must use IEEE representation.

template<int bits>
    alias using fast_10ieeefloat
```

§ 8.7
The alias `fast_10ieeefloat` refers to the fastest built-in decimal floating-point type of at least `bits` bits. If there are two types of the same size, it refers to the type that is closest to double in promotion order. The type must use IEEE representation.

```template<int bits> alias using least_10ieeefloat```

The alias `least_10ieeefloat` refers to the smallest built-in decimal floating-point type of at least `bits` bits. If there are two types of the same size, it refers to the type that is closest to double in promotion order. The type must use IEEE representation.
There's no default constructor in `integer_data_proxy`. This makes the class very hard to use.
The class describes an object that can be used to examine and modify the internal representation of an object of type `integer`. This allows advanced users to portably implement algorithms that are not provided natively.

Looks like the requirement "There can be only one `integer_data_proxy` object" may add overhead to the implementation. Implementation has to refcount the active `integer_data_proxy` to assert that there’s only one `integer_data_proxy`. This could be possibly avoided by adding a `integer_data_proxy(integer&& rhs)` constructor.

There can be only one `integer_data_proxy` object associated with a particular `integer` object at any given time; that object is obtained by calling the `get_data_proxy` member function on the `integer` object. The resulting object can be moved but not copied.

typedef unspecified arithmetic_type;

The typedef defines a synonym for a signed arithmetic type that is large enough to hold the product of the largest values that the implementation will store in an object of type data_type.

iterator begin();

*Returns:* An iterator object such that the iterators `[begin(), end())` point to the internal data elements of the `integer` object.

size_t capacity() const noexcept;

*Returns:* The number of decimal digits that the `integer` object can represent without reallocating its internal storage.

const_iterator cbegin() const;

*Returns:* An iterator object such that the iterator range `[cbegin(), cend())` points to the internal data elements of the `integer` object.

const_iterator cend() const;

*Returns:* An iterator object such that the iterator range `[cbegin(), cend())` points to the internal data elements of the `integer` object.

typedef unspecified const_iterator;

The typedef defines a synonym for an iterator that can be used to access but not modify internal data elements of the `integer` object.

typedef unspecified const_reverse_iterator;

The typedef defines a synonym for a reverse iterator that can be used to access but not modify internal data elements of the `integer` object.

const_reverse_iterator crbegin() const;

*Returns:* The member function returns a reverse iterator object such that the iterator range `[crbegin(), crend())` points to the internal data elements of the `integer` object in reverse order.

const_reverse_iterator crend() const;

*Returns:* A reverse iterator object such that the iterator range `[crbegin(), crend())` points to the internal data elements of the `integer` object in reverse order.

typedef unspecified data_type;
The typedef defines a synonym for the type of the `integer` object's internal data elements.

```cpp
iterator end();
```

*Returns:* An iterator object such that the iterator range `[begin(), end())` points to the internal data elements of the `integer` object.

```cpp
integer_data_proxy(const integer_data_proxy&) = delete;
```

The copy constructor is deleted.

```cpp
integer_data_proxy(integer_data_proxy&& rhs);
```

*Effects:* Copies the contents of `rhs` and leaves `rhs` in an unspecified valid state.

```cpp
typedef unspecified iterator;
```

The typedef defines a synonym for an iterator that can be used to access internal data elements of the `integer` object.

```cpp
integer& operator=(const integer_data_proxy&) = delete;
integer& operator=(integer_data_proxy&&) = delete;
```

The copy assignment and move assignment operators are deleted.

```cpp
data_type operator[](size_t pos) const;
```

*Returns:* The value of the internal data element at index `pos`.

```cpp
data_type& operator[](size_t pos);
```

*Returns:* A reference to the internal data element at index `pos`.

```cpp
reverse_iterator rbegin();
```

*Returns:* A reverse iterator object such that the iterator range `[crbegin(), crend())` points to the internal data elements of the `integer` object in reverse order.

```cpp
reverse_iterator rend();
```

*Returns:* A reverse iterator object such that the iterator range `[crbegin(), crend())` points to the internal data elements of the `integer` object in reverse order.

```cpp
void reserve(size_t digits);
```

*Effects:* Ensures that `capacity() >= digits`.

```cpp
typedef unspecified reverse_iterator;
```

The typedef defines a synonym for a reverse iterator that can be used to access internal data elements of the `integer` object.

```cpp
void shrink_to_fit();
```

*Effect on original feature:* Is a non-binding request to reduce `capacity()` to hold the `integer` object's current stored value without wasted space.

```cpp
size_t size() const;
```

*Returns:* `capacity()`.

```cpp
typedef unspecified uarithmetic_type;
```

The typedef defines a synonym for an unsigned arithmetic type that is large enough to hold the product of the largest values that the implementation will store in an object of type `data_type`.

§ 9.1
9.2 Bits

namespace std {

    // 26.?? binary operations
    bits operator&(const bits& lhs, const bits& rhs);
    bits operator|(const bits& lhs, const bits& rhs);
    bits operator^(const bits& lhs, const bits& rhs);

    // 26.?? iostream specializations
    template<class CharT, class Traits>
    basic_ostream<CharT, Traits>& operator<<(basic_ostream<CharT, Traits>& os,
                                           const bits& val);

    template<class CharT, class Traits>
    basic_ostream<CharT, Traits>& operator>>(basic_ostream<CharT, Traits>& os,
                                           bits& val);
}

Class bits interferes with P0237 'Wording for fundamental bit manipulation utilities'. We should not duplicate efforts.

namespace std {
    class bits {
        public:

            class reference;

            // constructors
            bits() noexcept;
            template <class Ty>
            bits(Ty rhs) noexcept; // integral types only
            bits(initializer_list<uint_least32_t> list);

            template <class CharT, class Traits, class Alloc>
            explicit bits(const basic_string<CharT, Traits, Alloc>& str,
                           typename basic_string<CharT, Traits, Alloc>::size_t pos = 0,
                           typename basic_string<CharT, Traits, Alloc>::size_t count = std::basic_string<CharT>::npos,
                           CharT zero = CharT('0'),
                           CharT one = CharT('1'));

            template <class CharT>
            explicit bits(const CharT *ptr,
                           typename basic_string<CharT>::size_t count = std::basic_string<CharT>::npos,
                           CharT zero = CharT('0'),
                           CharT one = CharT('1'));

            explicit bits(const integer& val);
            explicit bits(integer&& val);

            bits(const bits& rhs);
            bits(bits&& rhs) noexcept;

§ 9.2
// assign and swap
template <class Ty>
  bits& operator=(Ty rhs); // integral types only
bits& operator=(const integer& rhs);
bits& operator=(integer&& rhs);
bits& operator=(const bits& rhs);
bits& operator=(bits&& rhs);
void swap(bits& rhs) noexcept;

// conversions
unsigned long to_ulong() const;
unsigned long long to_ullong() const;
template <class CharT = char, class Traits = char_traits<CharT>, class Alloc = allocator<CharT>>
  basic_string<CharT, Traits, Alloc> to_string(CharT zero = CharT('0'), CharT one = CharT('1')) const;

// logical operations
bits& operator&=(const bits& rhs);
bits& operator|=(const bits& rhs);
bits& operator^=(const bits& rhs);
bits operator~() const;
bits& operator<<=(size_t rhs);
bits& operator>>=(size_t rhs);
bits& operator<<(size_t rhs) const;
bits& operator>>(size_t rhs) const;

// element access and modification
bits& set() noexcept;
bits& set(size_t pos, bool val = true);
bits reset() noexcept;
bits reset(size_t pos);
bits& flip() noexcept;
bits& flip(size_t pos);
bool operator[](size_t pos) const;
reference operator[](size_t pos);
bool test(size_t pos) const noexcept;
bool all() const noexcept;
bool any() const noexcept;
bool none() const noexcept;
size_t count() const noexcept;
size_t count_not_set() const noexcept;

// comparison
bool operator==(const bits& rhs) const noexcept;
bool operator!=(const bits& rhs) const noexcept;

// capacity
size_t size() const noexcept;
size_t capacity() const noexcept;
void reserve(size_t bit_count);
void shrink_to_fit();
}; // namespace std
The class describes an object that represents an unbounded set of bits.

9.2.1 Constructors

bits() noexcept;

*Effects:* Constructs an object whose value is 0.

```cpp
template <class Ty>
bits(Ty rhs) noexcept;
```

*Effects:* Constructs an object whose value is the ones-complement representation of `rhs`. Shall not take part in overload resolution unless the type `Ty` is an integral type.

```cpp
bits(initializer_list<uint_least32_t> list);
```

*Effects:* Constructs an object whose stored value is equal to the elements of the `initializer_list` treated as a series of unsigned 32-bit digits with the leftmost digit being most significant. For example, the `initializer_list` 0xFE, 0xF0, 0xAA, 0x31 represents the value 0xFE * 323 + 0xF0 * 322 + 0xAA * 321 + 0x31 * 320.

```cpp
template <class CharT, class Traits, class Alloc>
explicit bits::bits(const basic_string<CharT, Traits, Alloc>& str,
    typename basic_string<CharT, Traits, Alloc>::size_t pos = 0,
    typename basic_string<CharT, Traits, Alloc>::size_t count = basic_string<CharT>::npos,
    CharT zero = CharT('0'),
    CharT one = CharT('1'));
```

*Effects:* Construct an object whose value is the value represented by their argument, treating zero as 0 and one as 1.

```cpp
explicit bits(const integer& rhs);
explicit bits(integer&& rhs);
```

*Effects:* Construct objects whose value is the ones-complement representation of `rhs`.

```cpp
bits(const bits& rhs);
bits(bits&& rhs) noexcept;
```

*Effects:* Construct objects with the same value as `rhs`. The move constructor leaves `rhs` in an unspecified valid state.

9.2.2 Operations

```cpp
size_t capacity() const noexcept;
```

*Returns:* The number of bits that the object can represent without reallocating its internal storage.

```cpp
size_t count() const noexcept;
```

*Returns:* The number of bits in `*this` that are set, or `static_cast<size_t>(-1)` if the number of bits that are set is too large to fit in an object of type `size_t`.

```cpp
size_t count_not_set() const noexcept;
```
Returns: The number of bits in *this that are not set, or static_cast<size_t>(-1) if the number of bits that are not set is too large to fit in an object of type size_t.

void flip() const noexcept;
   
   Effects: Toggles all the bits in the stored value.

void flip(size_t pos);
   
   Effects: Toggles the bit at position pos in the stored value.

bool none() const noexcept;
   
   Returns: True only if none of the bits in *this is set.

void reserve(size_t bit_count);
   
   Effects: Ensures that capacity() >= bit_count.

bits& reset() noexcept;
   
   Effects: Clears all the bits of *this.
   
   Returns: *this.

bits& reset(size_t pos);
   
   Effects: Clears the bit as position pos.
   
   Returns: *this.

void set() noexcept;
   
   Effects: Sets all the bits of *this.
   
   Returns: *this.

void set(size_t pos, bool val = true);
   
   Effects: Sets the bit at position pos in the stored value to val.
   
   Returns: *this.

void shrink_to_fit();
   
   Effects: Is a non-binding request to reduce capacity() to hold the current stored value without wasted space.

size_t size() const noexcept;
   
   Returns: capacity().

bool test(size_t pos) const noexcept;
   
   Returns: True only if the bit at position pos in the stored value is non-zero.
9.2.3 Bits conversion  

```cpp
template <class CharT = char, class Traits = char_traits<CharT>, class Alloc = allocator<CharT>>
basic_string<CharT, Traits, Alloc> to_string(CharT zero = CharT('0'),
CharT one = CharT('1'));
```

*Returns:* A string representation of the bits in the value stored in *this*, using zero to represent 0 and one to represent 1.

```cpp
unsigned long long to_ullong() const;
```

*Returns:* A value equal to the stored value of *this*. It throws an exception of type `range_error` if the value cannot be represented as an `unsigned long long`.

```cpp
unsigned long to_ulong() const;
```

*Returns:* A value equal to the stored value of *this*. It throws an exception of type `range_error` if the value cannot be represented as a `long long`.

9.2.4 Bits operators

```cpp
template <class Ty>
bits& operator=(Ty rhs); // integral types only
```

*Effects:* Shall not take part in overload resolution unless the type Ty is an arithmetic type. The operator effectively executes `*this = integer(rhs)`.

*Returns:* *this.

```cpp
bits& operator=(const bits& rhs);
bits& operator=(bits&& rhs);
```

*Effects:* Store the value of rhs into *this.

*Returns:* *this.

```cpp
bits& operator=(const integer& rhs);
bits& operator=(integer&& rhs);
```

*Effects:* Store the ones-complement representation of rhs into *this.

*Returns:* *this.

```cpp
bool operator==(const bits& rhs) const noexcept;
```

*Returns:* True only if the stored value in *this is the same as the stored value in rhs.

```cpp
bool operator!=(const bits& rhs) const noexcept;
```

*Returns:* !(*this == rhs).

```cpp
bits operator&(const bits& lhs, const bits& rhs);
```

*Returns:* An object whose value is the bitwise AND of the values of lhs and rhs.

```cpp
bits& operator&=(const bits& rhs);
```

*Effects:* Sets the value of *this to the bitwise AND of the values of *this and rhs.

*Returns:* A reference to *this.

```cpp
bits operator|(const bits& lhs, const bits& rhs);
```

*Effects:* Shall not take part in overload resolution unless the type Ty is an arithmetic type. The operator effectively executes `*this = integer(rhs)`.

*Returns:* *this.
Returns: An object whose value is the bitwise inclusive OR of the values of `lhs` and `rhs`.

```cpp
bits& operator|=(const bits& rhs);
```

**Effects:** Sets the value of `*this` to the bitwise inclusive OR of the values of `*this` and `rhs`.

**Returns:** `*this`.

```cpp
bits operator~(const bits& lhs, const bits& rhs);
```

**Returns:** An object whose value is the bitwise exclusive OR of the values of `lhs` and `rhs`.

```cpp
bits& operator^=(const bits& rhs);
```

**Effects:** Sets the value of `*this` to the bitwise exclusive OR of the values of `*this` and `rhs`.

**Returns:** `*this`.

```cpp
bits operator~() const;
```

**Returns:** An object that holds the complement of the set of bits held by `*this`.

```cpp
bits operator>>(const bits& lhs, size_t rhs);
```

**Returns:** An object whose stored value is the value of the bits in `lhs` shifted right `rhs` positions.

```cpp
bits& operator>>=(size_t rhs);
```

**Effects:** Sets the stored value in `*this` to the value of the bits in `*this` shifted right `rhs` positions.

**Returns:** `*this`.

```cpp
template <class CharT, class Traits>
basic_istream<CharT, Traits>& operator>>(basic_istream<CharT, Traits>& is, bits& val);
```

**Effects:** Has the effect of `std::string temp; is >> temp; val = temp;`.

**Returns:** `is`.

```cpp
bits operator<<(const bits& lhs, size_t rhs);
```

**Returns:** An object whose stored value is the value of the bits in `lhs` shifted left `rhs` positions.

```cpp
bits& operator<<=(size_t rhs);
```

**Effects:** Sets the stored value in `*this` to the value of the bits in `*this` shifted left `rhs` positions.

**Returns:** `*this`.

```cpp
template <class CharT, class Traits>
basic_ostream<CharT, Traits>& operator<<(basic_ostream<CharT, Traits>& os, const bits& val);
```

**Effects:** Has the effect of `os << val.to_string()`.

**Returns:** `os`.

```cpp
bool operator[](size_t pos) const;
```

**Returns:** The value of the bit at position `pos`.

```cpp
reference operator[](size_t pos);
```

**Returns:** An object of type `bits::reference` that refers to the bit at position `pos`.

§ 9.2.4
9.2.5 bits::reference class

namespace std {
    class bits {
        class reference {
            public:
                reference& operator=(bool val) noexcept;
                reference& operator=(const reference& rhs) noexcept;
                bool operator~() const noexcept;
                operator bool() const noexcept;
                reference& flip() noexcept;
            }
        }
    }
}

The nested class bits::reference describes an object that can be used to manage a particular bit in an object of type bits.

reference& flip() noexcept;

Effects: Toggles the bit that the object manages.

reference& operator=(bool rhs) noexcept;

Effects: Sets the bit that the object manages to the value of rhs.

reference& operator=(const reference& rhs) noexcept;

Effects: Sets the bit that the object manages to the value managed by rhs.

bool operator~() const noexcept;

Returns: True if the bit managed by the object is set, otherwise false.

operator bool() const noexcept;

Returns: True if the bit that the object manages is set.

9.3 Integer

Default constructor of the class should be marked with constexpr so that the compiler could statically initialize it.

namespace std {
    class integer {
        public:
            // constructors
            constexpr integer() noexcept;

            template <class Ty>
            integer(Ty rhs) noexcept; // arithmetic types only

            integer(initializer_list<uint_least32_t> init);

            template <class CharT, class Traits, class Alloc>
            explicit integer(const basic_string<CharT, Traits, Alloc>& str);

            explicit integer(const bits& rhs);
            explicit integer(bits&& rhs);

§ 9.3
integer(const integer& rhs);
integer(integer&& rhs) noexcept;

// assign and swap
template <class Ty>
  integer& operator=(Ty rhs);  // arithmetic types only
integer& operator=(const bits& rhs);
integer& operator=(bits&& rhs);
integer& operator=(const integer& rhs);
integer& operator=(integer&& rhs);
void swap(integer& rhs) noexcept;

// conversions
explicit operator long long() const;
explicit operator unsigned long long() const;
explicit operator long double() const noexcept;
explicit operator bool() const noexcept;

// comparisons
int compare(const integer& rhs) const noexcept;

// arithmetic operations
integer& operator+=(const integer& rhs);
integer& operator-=(const integer& rhs);
integer& operator*=(const integer& rhs);
integer& operator/=(const integer& rhs);
integer& operator%=(const integer& rhs);
integer& operator++();
integer operator++(int);
integer& operator--();
integer operator--(int);
integer div(const integer& rhs);

integer& abs() noexcept;
integer& negate() noexcept;
integer operator+() const noexcept;
integer operator-() const noexcept;

integer& operator<<=(size_t rhs);
integer& operator>>=(size_t rhs);

// numeric operations
integer& sqr();
integer& sqrt();
integer& pow(const integer& exp);
integer& mod(const integer& rhs);
integer& mulmod(const integer& rhs, const integer& m);
integer& powmod(const integer& exp, const integer& m);

// observers
bool is_zero() const noexcept;
bool is_odd() const noexcept;
// accessors
integer_data_proxy get_data_proxy();

// capacity
size_t size() const noexcept;
size_t capacity() const noexcept;
void reserve(size_t digits);
void shrink_to_fit();

// namespace std
void swap(integer& lhs, integer& rhs) noexcept;

// comparisons
bool operator==(const integer& lhs, const integer& rhs) noexcept;
bool operator!=(const integer& lhs, const integer& rhs) noexcept;
bool operator<(const integer& lhs, const integer& rhs) noexcept;
bool operator<=(const integer& lhs, const integer& rhs) noexcept;
bool operator>(const integer& lhs, const integer& rhs) noexcept;
bool operator>=(const integer& lhs, const integer& rhs) noexcept;

// arithmetic operations
integer operator+(const integer& lhs, const integer& rhs);
integer operator-(const integer& lhs, const integer& rhs);
integer operator*(const integer& lhs, const integer& rhs);
integer operator/(const integer& lhs, const integer& rhs);
integer operator%(const integer& lhs, const integer& rhs);
pair<integer, integer> div(const integer& lhs, const integer& rhs);
integer abs(const integer& val);
integer operator<<(const integer& lhs, size_t rhs);
integer operator>>(const integer& lhs, size_t rhs);

// numeric operations
integer sqr(const integer& val);
integer sqrt(const integer& val);
integer pow(const integer& val, const integer& exp);
integer mod(const integer& lhs, const integer& rhs);
integer mulmod(const integer& lhs, const integer& rhs, const integer& m);
integer powmod(const integer& lhs, const integer& rhs, const integer& m);

integer gcd(const integer& a, const integer& b);
integer lcm(const integer& a, const integer& b);

// conversions
string to_string(const integer& val, int radix = 10);

// I/O operations
template <class CharT, class Traits>
basic_ostream<CharT, Traits>& operator<<(basic_ostream<CharT, Traits>& str,
const integer& val);

template <class CharT, class Traits>
basic_istream<CharT, Traits>& operator>>(basic_istream<CharT, Traits>& str,
    integer& val);

The class describes an object that manages an unbounded-precision signed integral type that can be used in
most contexts where an int could be used.
Any function specified to return an object of type integer may return an object of another type, provided
all the const member functions of the class integer are also applicable to that type.

integer abs(const integer& other);

    Returns: An object that holds the absolute value of other.

integer& abs() noexcept;

    Effects: Sets the stored value of *this to its absolute value and returns *this.

size_t capacity() const noexcept;

    Returns: The number of decimal digits that the object can represent without reallocating its internal
    storage.

int compare(const integer& rhs) const noexcept;

    Returns: A value less than 0 if *this is less than rhs, 0 if *this is equal to rhs, and greater than 0 if
    *this is greater than rhs.

pair<integer, integer> div(const integer& lhs, const integer& rhs);

    Returns: An object that is an instantiation of pair; its first field holds the quotient, lhs / rhs, and
    its second field holds the remainder, lhs % rhs.

integer div(const integer& rhs) const;

    Returns: The remainder, *this % rhs, and stores the quotient, *this / rhs, into *this.

integer gcd(const integer& a, const integer& b);

    Returns: An object whose value is the greatest common denominator of a and b.

Should we also provide the get_data_proxy() functionality for wide_integer?

integer_data_proxy get_data_proxy();

    Returns: An object of type integer_data_proxy that can be used to examine and modify the internal
    storage of *this. If an object of type integer_data_proxy that refers to *this exists at the
time of a call to this function, the function throws an exception object of type std::runtime_error.

9.3.1 Constructors

integer() noexcept;

    Effects: Constructs an object whose value is 0.

template <class Ty>
    integer(Ty val) noexcept; // arithmetic types only

    Effects: For integral types the constructor constructs an object whose value is val. For floating-point
types the constructor constructs an object whose value is the value of val with any fractional part
discarded. Shall not take part in overload resolution unless the type Ty is an arithmetic type.
integer(initializer_list<unspecified> list);

**Effects:** Constructs an object whose stored value is equal to the elements of the `initializer_list` treated as a series of unsigned 32-bit digits with the leftmost digit being most significant. For example, the `initializer list` 0xFE, 0xF0, 0xAA, 0x31 represents the value 0xFE * 32^3 + 0xF0 * 32^2 + 0xAA * 32^1 + 0x31 * 32^0.

```cpp
template<class CharT, class Traits, class Alloc>
extPLICIT integer(const basic_string<CharT, Traits, Alloc>& str);
```

**Effects:** Constructs an object whose value is the value represented by the `string` object. The `string` object shall have the form required for the `string` argument to the function `strtol` with a radix of base, and shall be interpreted as if by `strtol(str.c_str(), 0, base)`, except that the resulting value can never be outside the range of representable values.

```cpp
integer(const bits& rhs);
integer(bits&& rhs);
```

**Effects:** Construct an object whose stored value is the value in the bit pattern in `rhs` interpreted as a ones-complement representation of an integer value.

```cpp
integer(const integer& rhs);
integer(integer&& rhs) noexcept;
```

**Effects:** Construct objects with the same value as `rhs`. The move constructor leaves `rhs` in an unspecified valid state.

### 9.3.2 Operations

#### [numeric.integer.ops]

```cpp
bool is_odd() const noexcept;
```

**Returns:** True only if the stored value represents an odd number.

```cpp
bool is_zero() const noexcept;
```

**Returns:** True only if the stored value is zero.

```cpp
integer lcm(const integer& a, const integer& b);
```

**Returns:** An object whose value is the least common multiple of `a` and `b`.

```cpp
integer mod(const integer& lhs, const integer& rhs);
```

**Returns:** An object whose value is `lhs mod rhs`.

```cpp
integer& mod(const integer& rhs);
```

**Effects:** Sets the stored value in `*this` to `*this mod rhs`.

**Returns:** `*this`.

```cpp
integer& mulmod(const integer& lhs, const integer& rhs, const integer& m);
```

**Returns:** An object whose value is `(lhs * rhs) mod m`.

```cpp
integer& mulmod(const integer& rhs, const integer& m);
```

**Effects:** Sets the value of `*this` to `(*this * rhs) mod m`.

**Returns:** `*this`.
integer& negate() noexcept;
   Effects: Sets the stored value of *this to the negation of its previous value.
   Returns: this.

integer pow(const integer& val, const integer& exp);
   Requires: 0 <= exp.
   Returns: An object whose value is \( val^{exp} \).

integer& pow(const integer& exp);
   Requires: 0 <= exp.
   Effects: Sets the value of *this to \( *this^{exp} \).
   Returns: *this.

integer powmod(const integer& val, const integer& exp, const integer& m);
   Requires: 0 <= exp and \( m \neq 0 \).
   Returns: An object whose value is \( val^{exp} \mod m \).

integer& powmod(const integer& exp, const integer& m);
   Requires: 0 <= exp and \( m \neq 0 \).
   Effects: Sets the value of *this to \( *this^{exp} \mod m \).
   Returns: *this.

void reserve(size_t digits);
   Effects: Ensures that capacity() >= digits.

void shrink_to_fit();
   Effects: A non-binding request to reduce capacity() to hold the current stored value without wasted space.

size_t size() const noexcept;
   Returns: capacity().

integer sqrt(const integer& val);
   Requires: 0 <= val.
   Returns: An object whose value is the square root of the value held by val, discarding any fractional part.

integer& sqrt();

§ 9.3.2
Requires: 0 <= *this.

Effects: Sets the value of *this to the square root of the value held by *this, discarding any fractional part.

Returns: *this.

void swap(integer& lhs, integer& rhs) noexcept;

Effects: Swaps the stored values of lhs and rhs.

void swap(integer& rhs) noexcept;

Effects: Swaps the stored values of *this and rhs.

to_string usually does not have a radix argument. Probably a to_chars overload should be used instead.

string to_string(const integer& val, int radix = 10) const;

Returns: A string representation of the value stored in val, using radix as the radix.

9.3.3 Conversion [numeric.integer.conv]

explicit operator bool() const noexcept;

Returns: False only if *this is equal to 0.

explicit operator long double() const noexcept;

Returns: A value equal to the stored value of *this. If the stored value is outside the range that can be represented by an object of type long double the returned value is positive or negative infinity, as appropriate.

explicit operator long long() const;

Returns: A value equal to the stored value of *this. If the stored value cannot be represented as a long long it throws an exception of type range_error.

explicit operator unsigned long long() const;

Returns: A value equal to the stored value of *this. If the stored value cannot be represented as an unsigned long long it throws an exception of type range_error.

9.3.4 Comparison [numeric.integer.comp]

Spaceship operator should be used.

bool operator==(const integer& lhs, const integer& rhs) noexcept;

Returns: True only if the value stored in lhs is equal to the value stored in rhs.

bool operator!=(const integer& lhs, const integer& rhs) noexcept;

Returns: !(lhs == rhs).

bool operator>(const integer& lhs, const integer& rhs) noexcept;

Returns: rhs < lhs.

bool operator>=(const integer& lhs, const integer& rhs) noexcept;

Returns: !(lhs < rhs).
bool operator<(const integer& lhs, const integer& rhs) noexcept;

*Returns:* True only if lhs.compare(rhs) returns -1.

bool operator<=(const integer& lhs, const integer& rhs) noexcept;

*Returns:* !(rhs < lhs).

### 9.3.5 Assignment

[numeric.integer.assign]

```cpp
template <class Ty>
integer& operator=(Ty rhs);  // arithmetic types only
```

*Effects:* Shall not take part in overload resolution unless the type Ty is an arithmetic type. The operator effectively executes *this = integer(rhs).

*Returns:* *this.

```cpp
integer& operator=(const integer& rhs);
integer& operator=(integer&& rhs);
```

*Effects:* Store the value of rhs into *this.

*Returns:* *this.

```cpp
integer& operator=(const bits& rhs);
integer& operator=(bits&& rhs);
```

*Effects:* Store the value of rhs, interpreted as a ones-complement representation of an integer value, into *this.

*Returns:* *this.

### 9.3.6 Arithmetic operations

[numeric.integer.arithmetic_ops]

```cpp
integer operator+(const integer& lhs, const integer& rhs);
```

*Returns:* An object whose value is the sum of the values of lhs and rhs.

```cpp
integer operator+() const noexcept;
```

*Returns:* A copy of *this.

```cpp
integer& operator+=(const integer& rhs);
```

*Effects:* Sets the stored value of *this to the sum of the values of *this and rhs.

*Returns:* A reference to *this.

```cpp
integer& operator++();
```

*Effects:* Set the value stored in *this to *this + 1.

*Returns:* *this.

```cpp
integer& operator++(int);
```

*Returns:* An object whose value is the value stored in *this prior to the increment.

```cpp
integer operator-(const integer& lhs, const integer& rhs)
```

*Returns:* An object whose value is the difference between the values of lhs and rhs.

```cpp
integer operator-() noexcept;
```
Returns: An object whose value is the negation of the value of *this.

integer& operator-=(const integer&);

Effects: Sets the stored value of *this to the difference between the values of *this and rhs.
Returns: *this.

integer operator--();

Effects: Set the value stored in *this to *this - 1.
Returns: *this.

integer operator--(int);

Effects: Set the value stored in *this to *this - 1.
Returns: An object whose value is the value stored in *this prior to the decrement.

integer operator*(const integer& lhs, const integer& rhs);

Returns: An object whose value is the product of the values of lhs and rhs.

integer& operator*=(const integer& rhs);

Effects: Sets the stored value of *this to the product of the values of *this and rhs.
Returns: A reference to *this.

integer operator/=(const integer& rhs);

Effects: Sets the stored value of *this to the quotient of the value of *this divided by the value of rhs, discarding any fractional part.

integer operator%(const integer&, const integer&);

Returns: An object whose value is the remainder of the value of lhs divided by the value of rhs. The
remainder is the value such that (lhs / rhs) * rhs + lhs % rhs is equal to lhs.

integer& integer::operator%=(const integer&);

Effects: Sets the stored value of *this to the remainder of *this divided by the value of rhs.
Returns: *this.

integer operator>>(const integer& val, size_t rhs);

Returns: An object whose value is val / 2^rhs.

integer& operator>>=(size_t rhs);

Effects: Sets the value of *this to *this / 2^rhs.
Returns: *this.
9.3.7 I/O

Wide_integer adds to_chars and from_chars overloads. We should probably do the same here. to_chars
overloads make the type usable with std::format functions.

```
template <class Elem, class Traits>
    basic_istream<Elem, Traits>& operator>>(basic_istream<Elem, Traits>& is, integer& val);

    Effects: Has the effect of std::string temp; is >> temp; val = integer(temp);
    Returns: is.

integer operator<<(const integer& val, size_t rhs);

    Returns: An object whose value is val * 2^{rhs}.

integer& integer::operator<<=(size_t rhs);

    Effects: Sets the value of *this to *this * 2^{rhs}.
    Returns: *this.

template <class Elem, class Traits>
    basic_ostream<Elem, Traits>& operator<<(basic_ostream<Elem, Traits>& os, const integer& val);

    Effects: Has the effect of os << to_string(val).
    Returns: os.
```

Specialization of hash for integer is missing.
10 Generalized Type Conversion
[generalized_type_conversion]

Conversion between arbitrary numeric types requires something more practical than implementing the full cross product of conversion possibilities.

To that end, we propose that each numeric type promotes to an unbound type in the same general category. For example, integers of fixed size would promote to an unbound integer. In this promotion, there can be no possibility of overflow or rounding. Each type also demotes from that type. The demotion may have both round and overflow.

The general template conversion algorithm from type $S$ to type $T$ is to:

- Promote $S$ to its unbound type $S'$.
- Convert $S'$ to unbound type $T'$ of $T$.
- Demote $T'$ to $T$.

We expect common conversions to have specialized implementations.