C++ Numerics Work In Progress

Note: this is an early draft. It’s known to be incompletest 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 .................................................. 8

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

5 Feature test macros (Informative) ........................................... 12

6 Wide Integers ....................................................................... 13
  6.1 Class template `numeric_limits` .......................................... 13
  6.2 Header `<wide_integer>` synopsis ......................................... 13
  6.3 Template class `wide_integer` overview ................................ 15
  6.4 Specializations of `common_type` ........................................ 19
  6.5 Unary operators .............................................................. 20
  6.6 Binary operators ............................................................. 20
  6.7 Numeric conversions ........................................................ 21
  6.8 `iostream` specializations .................................................. 22
  6.9 Hash support ................................................................. 22

7 Rational math ....................................................................... 23
  7.1 Class `rational` ............................................................... 23

8 Parametric aliases .................................................................. 28
  8.1 Freestanding implementations ............................................ 28
  8.2 General ........................................................................... 28
  8.3 Header `<cstddef>` ........................................................... 28
  8.4 Parametric types ............................................................. 29
  8.5 Floating-point types ........................................................ 29
  8.6 Header `<cstfloat>` ........................................................... 29
  8.7 Parametric types ............................................................. 30

9 Unbounded Types ................................................................ 31
  9.1 Class `integer_data_proxy` ................................................ 31
9.2 Bits ................................................................. 34
9.3 Integer ............................................................ 40

10 Generalized Type Conversion 49
1 Foundations

1.1 Mission
The mission of this document is to collect different numbers related proposals in one to make it simpler to discover issues and to simplify editors work of porting the papers to \LaTeX (all the \LaTeX sources are available at https://github.com/ZaMaZaN4iK/numeric-papers-in-latex.git).

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.

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.
  
  — E.g. shift and add, multiply and add.

— Identify opportunities for new hardware.
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.
# Operations

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

<table>
<thead>
<tr>
<th>operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
</tr>
<tr>
<td>neg</td>
</tr>
<tr>
<td>add</td>
</tr>
<tr>
<td>sub</td>
</tr>
<tr>
<td>mul</td>
</tr>
<tr>
<td>rdiv</td>
</tr>
<tr>
<td>pdiv</td>
</tr>
<tr>
<td>quorem</td>
</tr>
<tr>
<td>mod</td>
</tr>
<tr>
<td>rem</td>
</tr>
<tr>
<td>scale/shift</td>
</tr>
<tr>
<td>ssu</td>
</tr>
<tr>
<td>ssd</td>
</tr>
<tr>
<td>dsu</td>
</tr>
<tr>
<td>dsd</td>
</tr>
<tr>
<td>lsh</td>
</tr>
<tr>
<td>ash</td>
</tr>
<tr>
<td>rsh</td>
</tr>
<tr>
<td>rtl</td>
</tr>
<tr>
<td>rtr</td>
</tr>
<tr>
<td>composite</td>
</tr>
<tr>
<td>add2</td>
</tr>
<tr>
<td>sub2</td>
</tr>
<tr>
<td>lshadd</td>
</tr>
<tr>
<td>lshsub</td>
</tr>
<tr>
<td>muladd</td>
</tr>
<tr>
<td>mulsub</td>
</tr>
<tr>
<td>muladd2</td>
</tr>
<tr>
<td>mulsub2</td>
</tr>
<tr>
<td>bitwise</td>
</tr>
<tr>
<td>not</td>
</tr>
<tr>
<td>and</td>
</tr>
<tr>
<td>xor</td>
</tr>
<tr>
<td>ior</td>
</tr>
<tr>
<td>dif</td>
</tr>
</tbody>
</table>

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

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

3.1 Overflow-Detecting Operations [overflow_detecting_ops]
The initial wording and motivation are available at P0103R0 "Overflow-Detecting and Double-Wide Arithmetic Operations". Was merged into this document because of the P0101R0 discussion at Kona 2015.

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.

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

3.2 Overflow-Handling Operations [overflow_handling_ops]
Overflow-handling operations require an overflow handling mode. We represent the mode in C++ as an enumeration:

```cpp
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 overflow(mode, T lower, T upper, U 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.

```cpp
template<typename T, typename U>  
constexpr T convert(overflow mode, U value);
```

Returns: overflow(mode, numeric_limits<T>::min(), numeric_limits<T>::max(), value).

§ 3.2
template<overflow Mode, typename T, typename U>
constexpr T convert(U value);

Returns: convert(Mode, value).

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

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

Returns: overflow(mode, lower, upper, value).

template<overflow Mode, typename T>
constexpr T limit(T lower, T upper, T value);

Returns: limit(Mode, lower, upper, value).

Common arguments can be elided with convenience functions.

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

Returns: limit(mode, 0, upper, value).

template<overflow Mode, typename T>
constexpr T limit_nonnegative(T upper, T value);

Returns: limit_nonnegative(Mode, upper, value).

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

Returns: limit(mode, -upper, upper, value).

template<overflow Mode, typename T>
constexpr T limit_signed(T upper, T value);

Returns: limit_signed(Mode, upper, value).

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

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

Returns: limit(mode, -upper-1, upper, value).

template<overflow Mode, typename T>
constexpr T limit_twoscomp(T upper, T value);

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.

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

Returns: overflow(mode, 0, 2^{upper} - 1, value).

template<overflow Mode, typename T>
constexpr T limit_nonnegative_bits(T upper, T value);

Returns: limit_nonnegative_bits(Mode, 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 Mode, typename T>
constexpr T limit_signed_bits(T upper, T value);

Returns: limit_signed_bits(Mode, 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 Mode, typename T>
constexpr T limit_twoscomp_bits(T upper, T value);

Returns: limit_twoscomp_bits(mode, upper, value).

Embedding overflow detection within regular operations can lead to enhanced performance. In particular, left shift is an 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 Mode, typename T>
constexpr T scale_up(T value, int count);

Returns: scale_up(Mode, value, count).

3.3 Rounding Operations [round_ops]
The initial wording and motivation are available at P0105R0 "Rounding and Overflow in C++".

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.

§ 3.3
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>
constexpr T convert(rounding mode, U value);
```

*Returns:* round(mode, U).

```
template<rounding Mode, typename T, typename U>
constexpr T convert(U value);
```

*Returns:* round<T>(Mode, U).

A division function has obvious utility.

```
template<typename T>
constexpr T divide(rounding mode, T dividend, T divisor);
```

*Returns:* is round(mode, dividend/divisor). Remember that division evaluates as a real number. Obviously, the implementation will use a different strategy, but it must yield the same result.

```
template<rounding Mode, typename T>
constexpr T divide(T dividend, T divisor);
```

*Returns:* divide(Mode, dividend, divisor).

Division by a power of two has substantial implementation efficiencies, and is used heavily in fixed-point arithmetic as a scaling mechanism. We represent the conjunction of these approaches with a rounding scale down (right shift).

```
template<typename T>
constexpr T scale_down(rounding mode, T value, int bits);
```

*Returns:* round(mode, dividend/2^bits).

```
template<rounding mode, typename T>
constexpr T scale_down(T value, int bits);
```

*Returns:* scale_down(mode, dividend, bits).

### 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.

```
template<typename T, typename U>
constexpr T convert(overflow omode, rounding rmode, U value);
```

*Returns:* overflow(omode, numeric_limits<T>::min(), numeric_limits<T>::max(), round(rmode, value)).

```
template<overflow OMode, rounding RMode, typename T, typename U>
constexpr T convert(U value);
```

*Returns:* convert(OMode, RMode; value).

Consider shifting as multiplication by a power of two. It has an analogy in a bidirectional shift, where a positive power is a left shift and a negative power is a right shift.
template<typename T>
constexpr T scale(overflow omode, rounding rmode, T value, int count);

Returns: count < 0 ? round(rmode, value*\(2^{count}\)) : overflow(omode, numeric_limits<T>::min(), numeric_limits<T>::max(), value*\(2^{count}\)).

template<overflow OMode, rounding RMode, typename T>
constexpr T scale(T value, int count);

Returns: scale(OMode, RMode value count).

3.5 Double-Word Operations

The initial wording and motivation are available at P0103R0 "Overflow-Detecting and Double-Wide Arithmetic 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>
<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 aliases as follows.

<table>
<thead>
<tr>
<th>alias name</th>
<th>result category</th>
<th>operation category</th>
</tr>
</thead>
<tbody>
<tr>
<td>largest_doubleWide_add</td>
<td>double-wide</td>
<td>add, add2, sub, sub2</td>
</tr>
<tr>
<td>largest_doubleWide_lsh</td>
<td>double-wide</td>
<td>lsh, lshadd</td>
</tr>
<tr>
<td>largest_doubleWide_mul</td>
<td>double-wide</td>
<td>mui, muladd, muladd2, mulsb, mulsb2</td>
</tr>
<tr>
<td>largest_doubleWide_div</td>
<td>double-wide</td>
<td>divn, divw, divnrem, divwrem</td>
</tr>
<tr>
<td>largest_doubleWide_all</td>
<td>double-wide</td>
<td>the minimum size of the four macros above</td>
</tr>
<tr>
<td>largest_singleWide_add</td>
<td>single-wide</td>
<td>add, add2, sub, sub2</td>
</tr>
<tr>
<td>largest_singleWide_lsh</td>
<td>single-wide</td>
<td>lsh, lshadd</td>
</tr>
<tr>
<td>largest_singleWide_mul</td>
<td>single-wide</td>
<td>mui, muladd, muladd2, mulsb, mulsb2</td>
</tr>
<tr>
<td>largest_singleWide_div</td>
<td>single-wide</td>
<td>divn, divw, divnrem, divwrem</td>
</tr>
<tr>
<td>largest_singleWide_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);
constexpr U split_lower(DS value);
constexpr DS wide_build(S upper, U lower);
```

```cpp
constexpr U split_upper(DU value);
constexpr U split_lower(DU value);
constexpr DU wide_build(U upper, U lower);
```

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

```cpp
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 addend1, 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_lshsub(S multiplicand, int count, S subtrahend);
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 addend1, 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_divwrem(S* remainder, DS dividend, S divisor);
```

```cpp
constexpr DU wide_lsh(U multiplicand, int count);
constexpr DU wide_add(U augend, U addend);
constexpr DU wide_sub(U minuend, U subtrahend);
constexpr DU wide_mul(U multiplicand, U multiplier);
constexpr DU wide_add2(U augend, U addend1, U addend2);
constexpr DU wide_sub2(U minuend, U subtrahend1, U subtrahend2);
constexpr DU wide_lshadd(U multiplicand, int count, U addend);
constexpr DU wide_lshsub(U multiplicand, int count, U subtrahend);
constexpr DU wide_muladd(U multiplicand, U multiplier, U addend);
constexpr DU wide_mulsub(U multiplicand, U multiplier, U subtrahend);
constexpr DU wide_muladd2(U multiplicand, U multiplier, U addend1, U addend2);
constexpr DU wide_mulsub2(U multiplicand, U multiplier, U subtrahend1, U subtrahend2);
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_divwrem(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.

```cpp
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 addend1, 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_lshsub(U* product, S multiplicand, int count, S subtrahend);
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_muladd2(U* product, U multiplicand, U multiplier, U addend1, U addend2);
constexpr U split_mulsub(U* product, U multiplicand, U multiplier, U subtrahend);
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]

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.

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

Effects: Add the word addend to the multiplicand of length length, leaving the result in the multiplicand.
Returns: Any carry out from the accumulator.
```

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

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.
```

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

Effects: Multiply the product of length length by the multiplier, leaving the result in the product.
Returns: Any carry out from the product.
```

```cpp
constexpr U unsigned_subarray_mul_word(U* product, U* multiplicand, int length, U multiplier);

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.
```

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

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.
5 Feature test macros (Informative) [feature.test]

Table 4 — Feature-test macro(s)

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>__cpp_lib_wide_integer</td>
<td>201912</td>
</tr>
<tr>
<td>__cpp_lib_integer</td>
<td>201912</td>
</tr>
<tr>
<td>__cpp_lib_numeric_aliases</td>
<td>201912</td>
</tr>
<tr>
<td>__cpp_lib_numeric_machine_layer</td>
<td>201912</td>
</tr>
<tr>
<td>__cpp_lib_rational</td>
<td>201912</td>
</tr>
</tbody>
</table>
6  Wide Integers

The initial wording and motivation are available at P0539R4 "A Proposal to add wide int Template Class".
Was merged into this document because of the SG6 discussion of the unpublished document and as the consequences of multiple P0539 discussions.

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]

6.2 Header <wide_integer> synopsis

```cpp
#include <compare>

namespace std {

    // 26.??.2 class template wide_integer
template<size_t Bits, typename S> class wide_integer;

    // 26.??.?? 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.??.?? 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.??.?? binary operations
    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>>
    operator*(const wide_integer<Bits, S>& val) noexcept;

```

§ 6.2
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(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(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) 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);

// 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);

// 26.?? 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);

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) noexcept;
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;
a *= 2.0;
a -= 12_int128;
assert(a > 0);
```

### 6.3 Template class `wide_integer` overview

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;
a *= 2.0;
a -= 12_int128;
assert(a > 0);
```
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.?? ?? 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.?? ?? 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&);
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&);
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&);

§ 6.3
The class template `wide_integer<Bits, 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 % (sizeof(int) * CHAR_BIT)` is equal to 0. `sizeof(wide_integer<Bits, unsigned>)` and `sizeof(wide_integer<Bits, signed>)` are required to be equal to `Bits * CHAR_BIT`.

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;
```

```cpp
#include <cassert>
#include <iostream>

#include <numeric>

int main()
{
    std::cout << std::numeric_limits<wide_integer<64, int>>::max() / 2 << std::endl;
}
```

### 6.3.1 `wide_integer` constructors

```cpp
constexpr wide_integer() noexcept = default;
```

*Effects:* Constructs an object with undefined value.

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

*Effects:* Constructs an object from `other` using the integral conversion rules.

```cpp
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.
6.3.2 wide_integer assignments

```cpp
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].

```cpp
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

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

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

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

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

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

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

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

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

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

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

```cpp
constexpr wide_integer<Bits, S>& operator++() noexcept(is_unsigned_v<S>);
```

```cpp
constexpr wide_integer<Bits, S> operator++(int) noexcept(is_unsigned_v<S>);
```

```cpp
constexpr wide_integer<Bits, S>& operator--() noexcept(is_unsigned_v<S>);
```

```cpp
constexpr wide_integer<Bits, S> operator--(int) noexcept(is_unsigned_v<S>);
```

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

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

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

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

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

*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&);
```

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

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

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

§ 6.3.3
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

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

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]

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

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]

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

Returns: value with inverted significant bits of val.

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

Returns: val *= -1 if S is true, otherwise the result is unspecified.

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

Returns: val.

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.

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);

Returns: CT(lhs) *= 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);

Returns: CT(lhs) /= 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(is_unsigned_v<S>);

Returns: CT(lhs) += 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(is_unsigned_v<S>);

Returns: CT(lhs) -= 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);

Returns: CT(lhs) %= rhs.

§6.6
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;

    Returns: CT(lhs) &= 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;

    Returns: CT(lhs) |= 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;

    Returns: CT(lhs) ^= 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);

    Returns: CT(lhs) <<= 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) noexcept;

    Returns: CT(lhs) >>= 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 strong_ordering operator<=>(const wide_integer<Bits, S>& lhs, const wide_integer<Bits2, S2>& rhs) noexcept;

    Returns: strong_ordering::equal if CT(lhs) - CT(rhs) == 0; otherwise returns strong_ordering::less if CT(lhs) - CT(rhs) < 0; otherwise strong_ordering::greater.

6.7 Numeric conversions

6.7.2 Numeric conversions

template<
    size_t Bits,
    typename S>
std::string to_string(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.

    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

[numeric.wide_integer.io]

```cpp
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`.

```cpp
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

[numeric.wide_integer.hash]

```cpp
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))`. 

§ 6.9 22
7 Rational math

The initial wording and motivation are available at N3611 "A Rational Number Library for C++".
Was merged into this document because of the SG6 discussion of the unpublished document and because of
the P0101R0 discussion at Kona 2015.

7.1 Class rational

```cpp
#include <compare>

class rational {
  public:
    constexpr rational() noexcept = default;
    rational(const rational& rat) = default;
    rational(rational&& rat) noexcept = default;
    rational& operator=(const rational& rat) = default;
    rational& operator=(rational&& rat) noexcept = default;
    rational() = default;

    explicit rational(float num);
    explicit rational(double num);
    explicit rational(long double num);
    explicit rational(integer num);

    rational(integer num, integer den);
    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);
```
```cpp
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);

private:
    integer numerator_; // exposition only
    integer denominator_; // exposition only
}

rational member functions:

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& 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;

§ 7.1
Returns: As if \(*this \neq 0\).

\[ \text{rational\& negate() noexcept;} \]

Effects: Changes the sign of \(*this\).
Returns: \(*this\).

\[ \text{rational\& invert() noexcept;} \]

Requires: the numerator is non-zero.
Effects: Swaps the numerator and denominator.
Returns: \(*this\).

### 7.1.1 rational member operators: ([rational.member.ops])

\[ \text{rational\& operator++();} \]

Effects: Adds 1 to \(*this\) and stores the result in \(*this\).
Returns: \(*this\).

\[ \text{rational\& operator--();} \]

Effects: Subtracts 1 from \(*this\) and stores the result in \(*this\).
Returns: \(*this\).

\[ \text{rational operator++(int);} \]

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

\[ \text{rational operator--(int);} \]

Effects: Subtracts 1 from \(*this\) and stores the result in \(*this\).
Returns: The value of \(*this\) before the subtraction.

\[ \text{rational\& operator+=(const integer\& rhs);} \]

Effects: Adds the integer value \( rhs \) to \(*this\) and stores the result in \(*this\).
Returns: \(*this\).

\[ \text{rational\& operator-=(const integer\& rhs);} \]

Effects: Subtracts the integer value \( rhs \) from \(*this\) and stores the result in \(*this\).
Returns: \(*this\).

\[ \text{rational\& operator*=(const integer\& rhs);} \]

Effects: Multiplies \(*this\) by the integer value \( rhs \) and stores the result in \(*this\).
Returns: \(*this\).

\[ \text{rational\& operator/=(const integer\& rhs);} \]

Requires: \( rhs \neq 0 \).
Effects: Divides \(*this\) by the integer value \( rhs \) and stores the result in \(*this\).
Returns: \(*this\).
rational& operator+=(const rational& rhs);
   
   **Effects:** Adds the rational value rhs to *this and stores the result in *this.
   
   **Returns:** *this.

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

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

rational& operator/=(const rational& rhs);
   
   **Requires:** rhs != 0.
   
   **Effects:** Divides *this by the rational value rhs and stores the result in *this.
   
   **Returns:** *this.

7.1.2 rational non-member operators: [rational.ops]

rational operator+(const rational& val);
   
   **Returns:** rational(val).

rational operator-(const rational& val);
   
   **Returns:** rational(val).negate().

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

rational operator-(const rational& lhs, const rational& rhs);
   
   **Returns:** rational(lhs) -= rhs.

rational operator*(const rational& lhs, const rational& rhs);
   
   **Returns:** rational(lhs) *= rhs.

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().

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

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

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

  Returns: As if after normalization lhs.numer() == rhs && lhs.denom() == 1.

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

  Returns: As if lhs.numer() <=> rhs * lhs.denom().
8 Parametric aliases

The initial wording and motivation are available at P0102R0 "C++ Parametric Number Type Aliases". Was merged into this document because of the P0101R0 discussion at Kona 2015.

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

8.1 Freestanding implementations

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

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>

Add the following entries to the synopsis before paragraph 1.

```cpp
namespace std {
    template<int bits> using exact_2int = implementation-defined;
    template<int bits> using fast_2int = implementation-defined;
    template<int bits> using least_2int = implementation-defined;
    template<int bits> using exact_2uint = implementation-defined;
    template<int bits> using fast_2uint = implementation-defined;
    template<int bits> using least_2uint = implementation-defined;
}
```

```cpp
inline constexpr size_t max_bits_2int = implementation-defined;
inline constexpr size_t max_bits_2uint = implementation-defined;
```

8.4 Parametric types

Add a new section with the following paragraphs.

The aliases below are conditionally supported. The max_bits_2int variable gives the largest integer size (in bits) supported by the aliases. [Note: All variants support the same sizes. â†œ End 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> 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> 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> 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> 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.

template<int bits> 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> 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.

namespace std {
  template<int bits> using exact_2ieeefloat = implementation-defined;
  template<int bits> using fast_2ieeefloat = implementation-defined;
  template<int bits> using least_2ieeefloat = implementation-defined;
  template<int bits> using exact_10ieeefloat = implementation-defined;
  template<int bits> using fast_10ieeefloat = implementation-defined;
  template<int bits> using least_10ieeefloat = implementation-defined;
}

inline constexpr size_t max_bits_2ieeefloat = implementation-defined;
inline constexpr size_t max_bits_10ieeefloat = implementation-defined;
8.7 Parametric types

Add a new section with the following paragraphs.

The aliases below are conditionally supported. The `max_bits_2ieeefloat` variables gives the largest binary floating-point size (in bits) supported by the aliases. [Note: All variants support the same sizes. â€”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> 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.

```cpp
template<int bits> 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.

```cpp
template<int bits> 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. â€”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> 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.

```cpp
template<int bits> using fast_10ieeefloat
```

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.

```cpp
template<int bits> 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.
9 Unbounded Types

The initial wording and motivation are available at N4038 "Proposal for Unbounded-Precision Integer Types". Was merged into this document because of the P0101R0 discussion at Kona 2015.

9.1 Class integer_data_proxy

namespace std {
    class integer_data_proxy {

        // type names
        typedef unspecified data_type;
        typedef unspecified arithmetic_type;
        typedef unspecified urithmetic_type;
        typedef unspecified iterator;
        typedef unspecified const_iterator;
        typedef unspecified reverse_iterator;
        typedef unspecified const_reverse_iterator;

        // constructors
        integer_data_proxy(const integer_data_proxy& rhs) = delete;
        integer_data_proxy(integer_data_proxy&& rhs);

        // assign
        integer_data_proxy& operator=(const integer_data_proxy& rhs) = delete;
        integer_data_proxy& operator=(integer_data_proxy&& rhs) = delete;

        // iterators
        iterator begin() noexcept;
        iterator end() noexcept;
        reverse_iterator rbegin() noexcept;
        reverse_iterator rend() noexcept;
        const_iterator cbegin() const noexcept;
        const_iterator cend() const noexcept;
        const_reverse_iterator crbegin() const noexcept;
        const_reverse_iterator crend() const noexcept;

        // element access
        data_type operator[](size_t pos) const;
        data_type& operator[](size_t pos);

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

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.
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.

`iterator end();`

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

`integer_data_proxy(const integer_data_proxy&) = delete;`
The copy constructor is deleted.

\texttt{integer\_data\_proxy(integer\_data\_proxy&& rhs)};

\textit{Effects:} Copies the contents of \texttt{rhs} and leaves \texttt{rhs} in an unspecified valid state.

typedef unspecified iterator;

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

\texttt{integer\& operator=(const integer\_data\_proxy\&)} = delete;
\texttt{integer\& operator=(integer\_data\_proxy\&\&)} = delete;

The copy assignment and move assignment operators are deleted.

\texttt{data\_type operator[] (size\_t pos) const};

\textit{Returns:} The value of the internal data element at index \texttt{pos}.

\texttt{data\_type\& operator[] (size\_t pos)};

\textit{Returns:} A reference to the internal data element at index \texttt{pos}.

\texttt{reverse\_iterator rbegin();}

\textit{Returns:} A reverse iterator object such that the iterator range \texttt{[rbegin(), rend())} points to the internal data elements of the \texttt{integer} object in reverse order.

\texttt{reverse\_iterator rend();}

\textit{Returns:} A reverse iterator object such that the iterator range \texttt{[rbegin(), rend())} points to the internal data elements of the \texttt{integer} object in reverse order.

\texttt{void reserve(size\_t digits);}  
\textit{Effects:} Ensures that \texttt{capacity()} \texttt{\geq} \texttt{digits}.

typedef unspecified reverse\_iterator;

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

\texttt{void shrink\_to\_fit();}

\textbf{Effect on original feature:} Is a non-binding request to reduce \texttt{capacity()} to hold the \texttt{integer} object’s current stored value without wasted space.

\texttt{size\_t size() const};

\textit{Returns:} \texttt{capacity()}.

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 \texttt{data\_type}.
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);
}

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;

        // assign and swap
        template <class Ty>
        bits& operator=(Ty rhs);              // integral types only

    }
}

§ 9.2
The class describes an object that represents an unbounded set of bits.

### 9.2.1 Constructors

```cpp
bits() noexcept;
```

---

**[numeric<bits.cons|numeric<bits.cons]**

---

§ 9.2.1
Effects: Constructs an object whose value is 0.

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

Effects: Constructs an object whose value is the one's-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 * 32^3 + 0xF0 * 32^2 + 0xAA * 32^1 + 0x31 * 32^0.

```cpp
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 = 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);
```

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

Effects: Construct objects whose value is the one's-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 [numeric<bits.ops]

```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<

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

Returns: The number of bits in *this that are not set, or static_cast<

```cpp
void flip() const noexcept;
```

§ 9.2.2
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 [numeric.bits.conv]

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.

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.

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

```c
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.
```

```c
bits& operator=(const bits& rhs);
    Effects: Store the value of rhs into *this.
    Returns: *this.
```

```c
bits& operator=(bits&& rhs);
    Effects: Store the ones-complement representation of rhs into *this.
    Returns: *this.
```

```c
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.
```

```c
bool operator!=(const bits& rhs) const noexcept;
    Returns: !(this == rhs).
```

```c
bits operator&(const bits& lhs, const bits& rhs);
    Returns: An object whose value is the bitwise AND of the values of lhs and rhs.
```

```c
bits& operator&=(const bits& rhs);
    Effects: Sets the value of *this to the bitwise AND of the values of *this and rhs.
    Returns: *this.
```

```c
bits operator|(const bits& lhs, const bits& rhs);
    Returns: An object whose value is the bitwise inclusive OR of the values of lhs and rhs.
```

```c
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.
```

```c
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.
```

```c
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.
```

```c
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.
```

```c
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.
```

```c
bits operator~() const;
    Returns: An object that holds the complement of the set of bits held by *this.
```
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.

    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.

    template <class CharT, class Traits>
    basic_istream<CharT, Traits>& operator>>(basic_istream<CharT, Traits>& is, bits& val);
    
    Effects: Has the effect of string temp; is >> temp; val = temp; .
    Returns: is.

    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.

    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.

    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.

bool operator[](size_t pos) const;

    Returns: The value of the bit at position pos.

reference operator[](size_t pos);

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

9.2.5 bits::reference class

    [numeric<bits::reference]

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;
            }; // class reference
    }; // class bits
} // namespace std

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

§ 9.2.5
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

#include <compare>

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);
       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;
   } // class integer
} // namespace std

§ 9.3 40
// 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;
strong_ordering operator<=>(const integer& lhs, const integer& rhs) noexcept;
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.

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 runtime_error.

9.3.1 Constructors

constexpr 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 * 323 + 0xF0 * 322 + 0xAA * 321 + 0x31 * 320.

template<class CharT, class Traits, class Alloc>
explicit 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.

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.

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]

bool is_odd() const noexcept;

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

bool is_zero() const noexcept;

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

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

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

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

*Returns*: An object whose value is \(\text{lhs mod rhs}\).

integer& mod(const integer& rhs);

*Effects*: Sets the stored value in \(*\text{this}\) to \(*\text{this mod rhs}\).

*Returns*: \(*\text{this}\).

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

*Returns*: An object whose value is \((\text{lhs * rhs}) \mod m\).

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

*Effects*: Sets the value of \(*\text{this}\) to \((\text{*this * rhs}) \mod m\).

*Returns*: \(*\text{this}\).

integer& negate() noexcept;

*Effects*: Sets the stored value of \(*\text{this}\) to the negation of its previous value.

*Returns*: \(*\text{this}\).

integer pow(const integer& val, const integer& exp);

*Requires*: \(0 \leq \text{exp}\).

*Returns*: An object whose value is \(\text{val}^{\text{exp}}\).

integer& pow(const integer& exp);

*Requires*: \(0 \leq \text{exp}\).

*Effects*: Sets the value of \(*\text{this}\) to \(*\text{this}^{\text{exp}}\).

*Returns*: \(*\text{this}\).

integer powmod(const integer& val, const integer& exp, const integer& m);

*Requires*: \(0 \leq \text{exp} \land m \neq 0\).

*Returns*: An object whose value is \(\text{val}^{\text{exp}} \mod m\).

integer& powmod(const integer& exp, const integer& m);

*Requires*: \(0 \leq \text{exp} \land m \neq 0\).

*Effects*: Sets the value of \(*\text{this}\) to \(*\text{this}^{\text{exp}} \mod m\).

*Returns*: \(*\text{this}\).
void reserve(size_t digits);
   \textit{Effects:} Ensures that capacity() \( \geq \) digits.

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

size_t size() const noexcept;
   \textit{Returns:} capacity().

integer sqr(const integer& val);
   \textit{Returns:} An object whose value is \( \text{val} \times \text{val} \).

integer& sqr();
   \textit{Effects:} Sets the value of \*this to \*this \* \*this.
   \textit{Returns:} \*this.

integer sqrt(const integer& val);
   \textit{Requires:} 0 \( \leq \) val.
   \textit{Returns:} An object whose value is the square root of the value held by \text{val}, discarding any fractional part.

integer& sqrt();
   \textit{Requires:} 0 \( \leq \) \*this.
   \textit{Effects:} Sets the value of \*this to the square root of the value held by \*this, discarding any fractional part.
   \textit{Returns:} \*this.

void swap(integer& lhs, integer& rhs) noexcept;
   \textit{Effects:} Swaps the stored values of \text{lhs} and \text{rhs}.

void swap(integer& rhs) noexcept;
   \textit{Effects:} Swaps the stored values of \*this and \text{rhs}.

string to_string(const integer& val, int radix = 10) const;
   \textit{Returns:} A string representation of the value stored in \text{val}, using \text{radix} as the radix.

9.3.3 Conversion \[\text{numeric.integer.conv}\]

explicit operator bool() const noexcept;
   \textit{Returns:} False only if \*this is equal to 0.

explicit operator long double() const noexcept;
   \textit{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]

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.

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

Returns: strong_ordering{lhs.compare(rhs)};.

9.3.5 Assignment [numeric.integer.assign]

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.

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

Effects: Store the value of rhs into *this.

Returns: *this.

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]

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

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

integer operator+() const noexcept;

Returns: A copy of *this.

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.

integer& operator++();

§ 9.3.6
Effects: Set the value stored in `*this` to `*this + 1`.

Returns: `*this`.

`integer operator++(int);`

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

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

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

`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& lhs, const integer& rhs);`

Returns: An object whose value is the quotient of the value of `lhs` divided by the value of `rhs`, discarding any fractional part.

`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.

Returns: `*this`.

`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&);`

§ 9.3.6
Effects: Sets the stored value of *this to the remainder of *this divided by the value of rhs.

Returns: *this.

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

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

type operator>>(size_t rhs);

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

Returns: *this.

§ 9.3.7  I/O

template <class Elem, class Traits>

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

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

Returns: is.

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

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

type 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.
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 its 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.