Document Number: Dxxxx
Date: 2019-09-07
Revises: Nxxxx
Reply to: Alexander Zaitsev
zamazan4ik@tut.by

Working Draft, C++ Extensions for Numerics

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

Contents

1 Scope [scope]

1 This document describes extensions to the C++ Standard Library. This document specifies requirements for implementations of an interface that computer programs written in the C++ programming language may use to perform operations related to numbers, such as work with unbounded numbers. This document is applicable to information technology systems that can perform operations with big numbers. This document is applicable only to vendors who wish to provide the interface it describes.

2 Normative references [references]

1 The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — ISO/IEC 14882:2017, *Programming languages — C++*

2 The programming language and library described in ISO/IEC 14882 is herein called the C++ Standard. References to clauses within the C++ Standard are written as "C++ 2017 [library]".

3 Unless otherwise specified, the whole of the C++ Standard's Library introduction (C++ 2017 [library]) is included into this document by reference.

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

1 These macros allow users to determine which version of this document is supported by the headers defined by the specification. All headers in this document shall define the `__cpp_lib_experimental_net` feature test macro in Table 1.

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

Table 1 — Feature-test macro(s)

Macro name	Value
`__cpp_lib_wide_integer`	201909

4 Wide_integer [wide_integer]

4.1 Class template numeric_limits [numeric.limits]

[Note to editor: 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]) - end note]

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*]

4.2 Header <wide_integer> synopsis [numeric.wide_integer.syn]

```
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>>
    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);

  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);

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

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

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

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

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

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

  // 26.??.?? numeric conversions
  template<size_t Bits, typename S> std::string to_string(const wide_integer<Bits, S>& val);
  template<size_t Bits, typename S> std::wstring to_wstring(const wide_integer<Bits, S>& val);

  // 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;

  // 26.??.?? hash support
  template<class T> struct hash;
  template<size_t Bits, typename S> struct hash<wide_integer<Bits, S>>;


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

  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);

  template <size_t Bits>
  using wide_int = wide_integer<Bits, signed>;

  template <size_t Bits>
  using wide_uint = wide_integer<Bits, unsigned>

  // optional literals
  inline namespace literals {
  inline namespace wide_int_literals {

  constexpr wide_int<128> operator "" _int128(const char*);
  constexpr wide_int<256> operator "" _int256(const char*);
  constexpr wide_int<512> operator "" _int512(const char*);
  constexpr wide_uint<128> operator "" _uint128(const char*);
  constexpr wide_uint<256> operator "" _uint256(const char*);
  constexpr wide_uint<512> operator "" _uint512(const char*);

  } // namespace wide_int_literals
  } // namespace literals

} // namespace std
```

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

Example:

```
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);
```


4.3 Template class `wide_integer` overview [numeric.wide_integer.overview]

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

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

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

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

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

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

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

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

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

    template<typename Integral>
```



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

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

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

    constexpr wide_integer<Bits, S>& operator++() noexcept(is_unsigned_v<S>);
    constexpr wide_integer<Bits, S> operator++(int) noexcept(is_unsigned_v<S>);
    constexpr wide_integer<Bits, S>& operator--() noexcept(is_unsigned_v<S>);
    constexpr wide_integer<Bits, S> operator--(int) noexcept(is_unsigned_v<S>);

    // 26.??.2.?? observers:
    template <typename Arithmetic> constexpr operator Arithmetic() const noexcept;
      constexpr explicit operator bool() const noexcept;
  private:
    byte data[Bits / CHAR_BITS]; // exposition only
  };
} // namespace std
```

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

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

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

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

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

Example:

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

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

4.3.1 `wide_integer` construsctors [numeric.wide_integer.cons]

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

Effects: Constructs an object with undefined value.

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

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


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

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

4.3.2 `wide_integer` assignments [numeric.wide_integer.assign]

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

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

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

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

4.3.3 `wide_integer` compound assignments [numeric.wide_integer.cassign]

```
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator*=(const wide_integer<Bits2, S2>&);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator/=(const wide_integer<Bits2, S2>&);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator+=(const wide_integer<Bits2, S2>&) noexcept(is_unsigned_v<S>);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator-=(const wide_integer<Bits2, S2>&) noexcept(is_unsigned_v<S>);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator%=(const wide_integer<Bits2, S2>&);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator&=(const wide_integer<Bits2, S2>&) noexcept;
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator|=(const wide_integer<Bits2, S2>&) noexcept;
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator^=(const wide_integer<Bits2, S2>&) noexcept;
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator<<=(const wide_integer<Bits2, S2>&);
template<size_t Bits2, typename S2>
  constexpr wide_integer<Bits, S>& operator>>=(const wide_integer<Bits2, S2>&) noexcept;
constexpr wide_integer<Bits, S>& operator++() noexcept(is_unsigned_v<S>);
constexpr wide_integer<Bits, S> operator++(int) noexcept(is_unsigned_v<S>);
constexpr wide_integer<Bits, S>& operator--() noexcept(is_unsigned_v<S>);
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.

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

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`.


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

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

4.3.4 `wide_integer` observers [numeric.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.

4.4 Specializations of common_type [numeric.wide_integer.traits.specializations]

```
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

4.5 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`.

4.6 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`.

```
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 value of `CT(lhs)` is less than the value of `CT(rhs)`.

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

Returns: `true` if value of `CT(lhs)` is greater than the value of `CT(rhs)`.

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

Returns: `true` if value of `CT(lhs)` is equal or less than the value of `CT(rhs)`.

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

Returns: `true` if value of `CT(lhs)` is equal or greater than the value of `CT(rhs)`.


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

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

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

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

4.7 Numeric conversions [numeric.wide_integer.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);
```

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

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

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

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

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

4.8 iostream specializations [numeric.wide_integer.io]

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

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

2 *Returns:* `os`.

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

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

4 *Returns:* `is`.

4.9 Hash support [numeric.wide_integer.hash]

```
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))`.

