Conceptless Random Number Generation
in C++0X, version 2

The primary purpose of this paper is to excise concepts from the wording of the standard library’s random number facility. In addition, we have herein provided wording to address all known related library issues (except for some that, after discussion, were closed by LWG as NAD). The usual color conventions are used throughout the text to denote the proposed changes, all of which are relative to Working Draft N3035.

Please note that the paper first proposes a small change to `random_shuffle`’s description. Thereafter, the paper focuses exclusively on subclause 26.5.

We thank our reviewers, and acknowledge the Fermilab Computing Division for sponsoring our participation in the C++ standardization effort.
template<class ForwardIterator>
    ForwardIterator rotate(ForwardIterator first, ForwardIterator middle,
                           ForwardIterator last);

template<class ForwardIterator, class OutputIterator>
    OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                               ForwardIterator last, OutputIterator result);

// 25.3.12, shuffling:
template<class RandomAccessIterator>
    void random_shuffle(RandomAccessIterator first,
                         RandomAccessIterator last);

template<class RandomAccessIterator, class RandomNumberGenerator>
    void random_shuffle(RandomAccessIterator first,
                         RandomAccessIterator last,
                         RandomNumberGenerator&& rand);

template<class RandomAccessIterator, class UniformRandomNumberGenerator>
    void random_shuffle(RandomAccessIterator first,
                         RandomAccessIterator last,
                         UniformRandomNumberGenerator& rand);

// 25.3.13, partitions:
template<class InputIterator, class Predicate>
    bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);

template<class ForwardIterator, class Predicate>
    ForwardIterator partition(ForwardIterator first,
                               ForwardIterator last,
                               Predicate pred);

template<class BidirectionalIterator, class Predicate>
    BidirectionalIterator stable_partition(BidirectionalIterator first,
                                             BidirectionalIterator last,
                                             Predicate pred);

template <class InputIterator, class OutputIterator1,
           class OutputIterator2, class Predicate>
    pair<OutputIterator1, OutputIterator2>
    partition_copy(InputIterator first, InputIterator last,
                   OutputIterator1 out_true, OutputIterator2 out_false,
                   Predicate pred);

template<class ForwardIterator, class Predicate>
    ForwardIterator partition_point(ForwardIterator first,
                                     ForwardIterator last,
                                     Predicate pred);

// 25.4, sorting and related operations:
// 25.4.1, sorting:
template<class RandomAccessIterator>
    void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
    void sort(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);

    void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>

template<class BidirectionalIterator, class OutputIterator>
OutputIterator
reverse_copy(BidirectionalIterator first,
            BidirectionalIterator last, OutputIterator result);

Effects: Copies the range \([\text{first}, \text{last})\) to the range \([\text{result}, \text{result}+(\text{last}-\text{first}))\) such that for any non-negative integer \(i < (\text{last} - \text{first})\) the following assignment takes place: \(*(\text{result} + (\text{last} - \text{first}) - i) = *(\text{first} + i)\).

Requires: The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result}+(\text{last}-\text{first}))\) shall not overlap.

Returns: \(\text{result} + (\text{last} - \text{first})\).

Complexity: Exactly \(\text{last} - \text{first}\) assignments.

25.3.11 Rotate

\[\text{alg.rotate}\]

template<class ForwardIterator>
ForwardIterator rotate(ForwardIterator first, ForwardIterator middle,
                       ForwardIterator last);

Effects: For each non-negative integer \(i < (\text{last} - \text{first})\), places the element from the position \(\text{first} + i\) into position \(\text{first} + (i + (\text{last} - \text{middle})) \% (\text{last} - \text{first})\).

Returns: \(\text{first} + (\text{last} - \text{middle})\).

Remarks: This is a left rotate.

Requires: \([\text{first}, \text{middle})\) and \([\text{middle}, \text{last})\) shall be valid ranges. The type of \(*\text{first}\) shall satisfy the Swappable requirements (37), the MoveConstructible requirements (Table 33), and the MoveAssignable requirements (Table 35).

Complexity: At most \(\text{last} - \text{first}\) swaps.

template<class ForwardIterator, class OutputIterator>
OutputIterator
rotate_copy(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last, OutputIterator result);

Effects: Copies the range \([\text{first}, \text{last})\) to the range \([\text{result}, \text{result} + (\text{last} - \text{first}))\) such that for each non-negative integer \(i < (\text{last} - \text{first})\) the following assignment takes place: \(*(\text{result} + i) = *(\text{first} + (i + (\text{middle} - \text{first})) \% (\text{last} - \text{first}))\).

Returns: \(\text{result} + (\text{last} - \text{first})\).

Requires: The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result} + (\text{last} - \text{first}))\) shall not overlap.

Complexity: Exactly \(\text{last} - \text{first}\) assignments.

25.3.12 Random shuffle

\[\text{alg.random.shuffle}\]

template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first,
                    RandomAccessIterator last);

template<class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle(RandomAccessIterator first,
                    RandomAccessIterator last,
                    RandomNumberGenerator&& rand);

§ 25.3.12 List of Figures 803


```cpp
template<class RandomAccessIterator, class UniformRandomNumberGenerator>
void random_shuffle(RandomAccessIterator first, 
                    RandomAccessIterator last, 
                    UniformRandomNumberGenerator& g);
```

1. **Effects:** Permutes the elements in the range `[first, last)` such that each possible permutation of those elements has equal probability of appearance.

2. **Requires:** The type of `*first` shall satisfy the Swappable requirements (Table 37). The random number generating function object `rand` shall have a return type that is convertible to `iterator_traits<RandomAccessIterator>::difference_type`, and the call `rand(n)` shall return a randomly chosen value in the interval `[0, n)` for `n > 0` of type `iterator_traits<RandomAccessIterator>::difference_type`. The function object `type UniformRandomNumberGenerator` shall meet the requirements of a uniform random number generator (26.5.1.3) type whose return type is convertible to `iterator_traits<RandomAccessIterator>::difference_type`. [Editor’s note: These adjustments are intended to resolve LWG Issue 1093.]

3. **Complexity:** Exactly `(last - first) - 1` swaps.

4. **Remarks:** To the extent that the implementation of these functions makes use of random numbers, the implementation shall use the following sources of randomness:

   - The underlying source of random numbers for the first form of the function is implementation-defined. An implementation may use the `rand` function from the standard C library.
   - In the second form of the function, the function object `rand` shall serve as the implementation’s source of randomness.
   - In the third (`shuffle`) form of the function, the object `g` shall serve as the implementation’s source of randomness.

### 25.3.13 Partitions

```cpp
template <class InputIterator, class Predicate>
bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);
```

1. **Requires:** `InputIterator`’s value type shall be convertible to `Predicate`’s argument type.

2. **Returns:** `true` if `[first, last)` is partitioned by `pred`, i.e. if all elements that satisfy `pred` appear before those that do not.

3. **Complexity:** Linear. At most `last - first` applications of `pred`.

### 25.3.13 List of Figures

804
26.4.9 Additional Overloads

The following function templates shall have additional overloads:

- `arg norm`
- `conj proj`
- `imag real`

The additional overloads shall be sufficient to ensure:
1. If the argument has type `long double`, then it is effectively cast to `complex<long double>`.
2. Otherwise, if the argument has type `double` or an integer type, then it is effectively cast to `complex<double>`.
3. Otherwise, if the argument has type `float`, then it is effectively cast to `complex<float>`.

Function template `pow` shall have additional overloads sufficient to ensure, for a call with at least one argument of type `complex<T>`:
1. If either argument has type `complex<long double>` or type `long double`, then both arguments are effectively cast to `complex<long double>`.
2. Otherwise, if either argument has type `complex<double>`, `double`, or an integer type, then both arguments are effectively cast to `complex<double>`.
3. Otherwise, if either argument has type `complex<float>` or `float`, then both arguments are effectively cast to `complex<float>`.

26.4.10 Header `<ccomplex>`

The header behaves as if it simply includes the header `<complex>`.

26.4.11 Header `<complex.h>`

The header behaves as if it includes the header `<ccomplex>`. [Note: `<complex.h>` does not promote any interface into the global namespace as there is no C interface to promote. —end note]

26.5 Random number generation

This subclause defines a facility for generating (pseudo-)random numbers.

In addition to a few utilities, four categories of entities are described: uniform random number generators, random number engines, random number engine adaptors, and random number distributions. These categorizations are applicable to types that satisfy the corresponding concept requirements, to objects instantiated from such types, and to templates producing such types when instantiated. [Note: These entities are specified in such a way as to permit the binding of any uniform random number generator object `e` as the argument to any random number distribution object `d`, thus producing a zero-argument function object such as given by `bind(d,e)`. —end note]

Each of the entities specified via this subclause has an associated arithmetic type (3.9.1) identified as `result_type`. With `T` as the `result_type` thus associated with such an entity, that entity is characterized:

- a) as `boolean` or equivalently as `boolean-valued`, if `T` is `bool`;
- b) otherwise as `integral` or equivalently as `integer-valued`, if `numeric_limits<T>::is_integer` is true;
- c) otherwise as `floating` or equivalently as `real-valued`.

If integer-valued, an entity may optionally be further characterized as `signed` or `unsigned`, according to `numeric_limits<T>::is_signed`.

Unless otherwise specified, all descriptions of calculations in this subclause use mathematical real numbers.

Throughout this subclause, the operators `bitand`, `bitor`, and `xor` denote the respective conventional bitwise operations. Further:

- a) the operator `rshift` denotes a bitwise right shift with zero-valued bits appearing in the high bits of the result, and
b) the operator \( lshift_w \) denotes a bitwise left shift with zero-valued bits appearing in the low bits of the result, and whose result is always taken modulo \( 2^w \).

[Editor’s note: The header \(<\text{random}>\) synopsis, formerly located here, has been moved and now follows the Requirements section below.]

26.5.1 Concepts and related requirements for random number generation

26.5.1.1 General requirements

Throughout this subclause 26.5, the effect of instantiating a template:

- a) that has a template type parameter named \( \text{Sseq} \) is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of seed sequence (26.5.1.2).
- b) that has a template type parameter named \( \text{URNG} \) is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of uniform random number generator (26.5.1.3).
- c) that has a template type parameter named \( \text{Engine} \) is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of random number engine (26.5.1.4).
- d) that has a template type parameter named \( \text{RealType} \) is undefined unless the corresponding template argument is cv-unqualified and is one of \( \text{float}, \text{double}, \) or \( \text{long double} \).
- e) that has a template type parameter named \( \text{IntType} \) is undefined unless the corresponding template argument is cv-unqualified and is one of \( \text{short}, \text{int}, \text{long}, \text{long long}, \text{unsigned short}, \text{unsigned int}, \text{unsigned long}, \text{or unsigned long long} \).
- f) that has a template type parameter named \( \text{UIntType} \) is undefined unless the corresponding template argument is cv-unqualified and is one of \( \text{unsigned short}, \text{unsigned int}, \text{unsigned long}, \text{or unsigned long long} \).

Throughout this subclause 26.5, phrases of the form “\( x \) is an iterator of a specific kind” shall be interpreted as equivalent to the more formal requirement that “\( x \) is a value of a type satisfying the requirements of the specified iterator type.”

Throughout this subclause 26.5, any constructor that can be called with a single argument and that satisfies a requirement specified in this subclause shall be declared \textit{explicit}.

26.5.1.2 Concept SeedSequence

A \textit{seed sequence} is an object that consumes a sequence of integer-valued data and produces a requested number of unsigned integer values \( i, 0 \leq i < 2^{32} \), based on the consumed data. \[\textit{Note: Such an object provides a mechanism to avoid replication of streams of random variates. This can be useful, for example, in applications requiring large numbers of random number engines. — end note}\]

```cpp
concept SeedSequence<
    typename S
> : Semiregular<S>, DefaultConstructible<S> {
    UnsignedIntegralLike result_type;
    requires IntegralType<result_type>
    && True<sizeof uint32_t <= sizeof result_type>;
    template<IntegralLike T>
    requires IntegralType<T>
    S::S(initializer_list<T> il);,
    template<InputIterator Iter>
    requires IntegralLike<Iter::value_type>
    && IntegralType<Iter::value_type>
    S::S(Iter begin, Iter end, size_t u = numeric_limits<Iter::value_type>::digits);,
    template<RandomAccessIterator Iter>
    requires UnsignedIntegralLike<Iter::value_type>
    && OutputIterator<Iter, const result_type&>
    && IntegralType<Iter::value_type>
}```

§ 26.5.1.2 Numerics library 833
&& True<sizeof uint32_t <= sizeof Iter::value_type>
    void generate(S& q, Iter begin, Iter end);

size_t size(const S& q);
    template<OutputIterator<auto, const result_type&> Iter>
        void param(const S& q, Iter dest);
    }

    template<IntegralLike T>
        requires IntegralType<T>
            S::S(initializer_list<T> il);
        2    Effects: Same as S::S(il.begin(), il.end()).

    template<InputIterator Iter>
        requires IntegralLike<Iter::value_type>
            && IntegralType<Iter::value_type>
                S::S(Iter begin, Iter end,
                size_t u = numeric_limits<Iter::value_type>::digits);
        3    Effects: Constructs a SeedSequence object having internal state that depends on some or all of the bits of the supplied sequence [begin, end).

    template<RandomAccessIterator Iter>
        requires UnsignedIntegralLike<Iter::value_type>
            && OutputIterator<Iter, const result_type&>
            && IntegralType<Iter::value_type>
            && True<sizeof uint32_t <= sizeof Iter::value_type>
                void generate(S& q, Iter begin, Iter end);
        4    Effects: Does nothing if begin == end. Otherwise, fills the supplied range [begin, end) with 32-bit quantities that depend on the sequence supplied to the constructor and possibly also on the history of generate's previous invocations.

size_t size(const S& q);
        5    Returns: The number of 32-bit units that would be returned by a call to param().

    template<OutputIterator<auto, const result_type&> Iter>
        void param(const S& q, Iter dest);
        6    Effects: Copies to the given destination a sequence of 32-bit units that can be provided to the constructor of a second object of the same type, and that would reproduce in that second object a state indistinguishable from the state of the first object.

[Editor's note: In the process of reformulating the above SeedSequence concept into the corresponding requirements table below, two errors were noted and corrected: First, the concept was wrong to require copyability (implied by Semiregular). The requirements table therefore imposes no such requirements as CopyConstructible or CopyAssignable. Second, the constructor taking a pair of input iterators should have had its third parameter, u, removed as part of the resolution of LWG Issue 803. The corresponding requirements table entry therefore does not mention this parameter.]

7 A class S satisfies the requirements of a seed sequence if the expressions shown in Table 109 are valid and have the indicated semantics, and if S also satisfies all other requirements of this section 26.5.1.2. In that Table and throughout this section:
   a) T is the type named by S's associated result_type;
b) q is a value of S and r is a possibly const value of S;
c) ib and ie are input iterators with an unsigned integer value_type of at least 32 bits;
d) rb and re are mutable random access iterators with an unsigned integer value_type of at least 32 bits;
e) ob is an output iterator; and
f) il is a value of initializer_list<T>.

Table 109 — Seed sequence requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>S::result_type</td>
<td>T</td>
<td>T is an unsigned integer type (3.9.1) of at least 32 bits.</td>
<td>compile-time</td>
</tr>
<tr>
<td>S()</td>
<td>—</td>
<td>Creates a seed sequence with the same initial state as all other default-constructed seed sequences of type S.</td>
<td>constant</td>
</tr>
<tr>
<td>S(ib,ie)</td>
<td>—</td>
<td>Creates a seed sequence having internal state that depends on some or all of the bits of the supplied sequence [ib,ie).</td>
<td>(\mathcal{O}(ie - ib))</td>
</tr>
<tr>
<td>S(il)</td>
<td>—</td>
<td>Same as S(il.begin(), il.end()).</td>
<td>same as S(il.begin(), il.end())</td>
</tr>
<tr>
<td>q.generate(rb,re)</td>
<td>void</td>
<td>Does nothing if rb == re. Otherwise, fills the supplied sequence [rb,re) with 32-bit quantities that depend on the sequence supplied to the constructor and possibly also depend on the history of generate's previous invocations.</td>
<td>(\mathcal{O}(re - rb))</td>
</tr>
<tr>
<td>r.size()</td>
<td>size_t</td>
<td>The number of 32-bit units that would be copied by a call to r.param.</td>
<td>constant</td>
</tr>
<tr>
<td>r.param(ob)</td>
<td>void</td>
<td>Copies to the given destination a sequence of 32-bit units that can be provided to the constructor of a second object of type S, and that would reproduce in that second object a state indistinguishable from the state of the first object.</td>
<td>(\mathcal{O}(r.size()))</td>
</tr>
</tbody>
</table>

26.5.1.3 Concept UniformRandomNumberGeneratorUniform random number generator requirements [rand.req.urng]

A uniform random number generator g of type G is a function object returning unsigned integral values such that each value in the range of possible results has (ideally) equal probability of being returned. [Note: The degree to which g’s results approximate the ideal is often determined statistically. — end note]
concept UniformRandomNumberGenerator<typename G> : Callable<G> {
    requires UnsignedIntegralLike<result_type>
    && IntegralType<result_type>;

    static constexpr result_type G::min();
    static constexpr result_type G::max();

    axiom NonemptyRange(G& g) {
        G::min() < G::max();
    }

    axiom InRange(G& g, result_type& r) {
        r = g(), G::min() <= r && r <= G::max();
    }
}

result_type operator()(G& g); // from Callable<G>

2    Complexity: amortized constant.

3 A class G satisfies the requirements of a uniform random number generator if the expressions shown in Table 110 are valid and have the indicated semantics, and if G also satisfies all other requirements of this section 26.5.1.3. In that Table and throughout this section:
   a) T is the type named by G’s associated result_type, and
   b) g is a value of G.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>G::result_type</td>
<td>T</td>
<td>T is an unsigned integer type (3.9.1).</td>
<td>compile-time</td>
</tr>
<tr>
<td>g()</td>
<td>T</td>
<td>Returns a value in the closed interval [G::min(), G::max()].</td>
<td>amortized constant</td>
</tr>
<tr>
<td>G::min()</td>
<td>T</td>
<td>Denotes the least value potentially returned by operator().</td>
<td>compile-time</td>
</tr>
<tr>
<td>G::max()</td>
<td>T</td>
<td>Denotes the greatest value potentially returned by operator().</td>
<td>compile-time</td>
</tr>
</tbody>
</table>

4 The following relation shall hold: G::min() < G::max().

### 26.5.1.4 Concept RandomNumberEngine
A random number engine (commonly shortened to engine) e of type E is a uniform random number generator that additionally meets the requirements (e.g., for seeding and for input/output) specified in this section.

2 Unless otherwise specified, the complexity of each function specified via the RandomNumberEngine concept (including those specified via any less-refined concept) shall be $O(size\ of\ state)$.

```cpp
concept RandomNumberEngine<typename E> : Regular<E>, UniformRandomNumberGenerator<E> {
    explicit E::E(result_type value);
    template<SeedSequence Sseq> explicit E::E(Sseq& q);
    void seed(E& e) { e = E(); }
    void seed(E& e, result_type s) { e = E(s); }
    template<SeedSequence Sseq> void seed(E& e, Sseq& q) { e = E(q); }
}
```
void discard(E& e, unsigned long long z) { for(; z != 0ULL; --z) e(); }

template<OutputStreamable OS> OS& operator<<(OS& os, const E& e);
template<InputStreamable IS> IS& operator>>(IS& is, E& e);

axiom Uniqueness( E& e, E& f, result_type s, Sseq& q) {
    (seed(e) , e) == (seed(f) , f);
    (seed(e,s), e) == (seed(f,s), f);
    (seed(e,q), e) == (seed(f,q), f);
}

axiom Seeding(E& e, result_type s, Sseq& q) {
    (seed(e) , e) == E();
    (seed(e,s), e) == E(s);
    (seed(e,q), e) == E(q);
}

3 At any given time, e has a state eᵢ for some integer i ≥ 0. Upon construction, e has an initial state e₀. An engine’s state may be established via a constructor, a seed function, assignment, or a suitable operator>>.

4 E’s specification shall define:
   a) the size of E’s state in multiples of the size of result_type, given as an integral constant expression;
   b) the transition algorithm TA by which eᵢ’s state eᵢ is advanced to its successor state eᵢ₊₁; and
   c) the generation algorithm GA by which an engine’s state is mapped to a value of type result_type.

bool operator==(const E& e1, const E& e2); // from Regular<E>

5 Returns: true if S₁ = S₂, where S₁ and S₂ are the infinite sequences of values that would be generated, respectively, by repeated future calls to e1() and e2(). Otherwise returns false.

result_type operator() (E& e); // from UniformRandomNumberGenerator<E>

6 Effects: Sets e’s state to eᵢ₊₁ = TA(eᵢ).
7 Returns: GA(eᵢ).
8 Complexity: as specified in 26.5.1.3 via the UniformRandomNumberGenerator concept.

explicit E::E(result_type s);

9 Effects: Creates an engine with an initial state that depends on s.

template<SeedSequence Sseq> explicit E::E(Sseq& q);

10 Effects: Creates an engine with an initial state that depends on a sequence produced by one call to SeedSequence::generate(q, ...).
11 Complexity: Same as complexity of SeedSequence::generate(q, ...) when called on a sequence whose length is size of state.
12 Note: This constructor (as well as the corresponding seed() function below) may be particularly useful to applications requiring a large number of independent random sequences.

void seed(E& e);

13 Postcondition: e == E().
14 Complexity: same as E().

§ 26.5.1.4 Numerics library 837
```c
void seed(E& e, result_type s);
15  Postcondition: e == E(s).
16  Complexity: same as E(s).

template<SeedSequence Sseq> void seed(E& e, Sseq& q);
17  Postcondition: e == E(q) (using the original value of q).
18  Complexity: same as E(q).

void discard(E& e, unsigned long long z);
19  Effects: Advances the engine's state from e_i to e_{i+z} by any means equivalent to the default implement-
20  ation specified above.
21  Complexity: no worse than the complexity of z consecutive calls to operator().
22  Note: This operation is common in user code, and can often be implemented in an engine-specific man-
23  ner so as to provide significant performance improvements over the default implementation specified
24  above.

template<OutputStreamable OS> OS&& operator<<(OS&& os, const E& e);
22  Effects: With os.fmtflags set to ios_base::dec|ios_base::left and the fill character set to the
23  space character, writes to os the textual representation of e's current state. In the output, adjacent
24  numbers are separated by one or more space characters.
25  Returns: the updated os.
26  Postcondition: The os.fmtflags and fill character are unchanged.

template<InputStreamable IS> IS&& operator>>(IS&& is, E& e);
25  Precondition: is provides a textual representation that was previously written using an output stream
26  whose imbued locale was the same as that of is, and whose associated types OutputStreamable::charT
27  and OutputStreamable::traits were respectively the same as those of is.
28  Effects: With is.fmtflags set to ios_base::dec, sets e's state as determined by reading its textual
29  representation from is. If bad input is encountered, ensures that e's state is unchanged by the
30  operation and calls is.setstate(ios::failbit) (which may throw ios::failure [27.5.4.3]). If a
31  textual representation written via os << x was subsequently read via is >> v, then x == v provided
32  that there have been no intervening invocations of x or of v.
33  Returns: the updated is.
34  Postcondition: The is.fmtflags are unchanged.

A class E that satisfies the requirements of a uniform random number generator (26.5.1.3) also satisfies
the requirements of a random number engine if the expressions shown in Table 111 are valid and have the
indicated semantics, and if E also satisfies all other requirements of this section 26.5.1.4. In that Table and
throughout this section:
  a) T is the type named by E’s associated result_type;
  b) e is a value of E, v is an lvalue of E, x and y are (possibly const) values of E;
  c) s is a value of T;
  d) q is an lvalue satisfying the requirements of a seed sequence (26.5.1.2);
  e) z is a value of type unsigned long long;
  f) os is an lvalue of the type of some class template specialization basic_ostream<charT, traits>; and
  g) is is an lvalue of the type of some class template specialization basic_istream<charT, traits>;
where charT and traits are constrained according to Clause 21 and Clause 27.
```
### Table 111 — Random number engine requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>E()</td>
<td>—</td>
<td>Creates an engine with the same initial state as all other default-constructed engines of type E.</td>
<td>(O) (size of state)</td>
</tr>
<tr>
<td>E(x)</td>
<td>—</td>
<td>Creates an engine that compares equal to x.</td>
<td>(O) (size of state)</td>
</tr>
<tr>
<td>E(s)</td>
<td>—</td>
<td>Creates an engine with initial state determined by s.</td>
<td>(O) (size of state)</td>
</tr>
<tr>
<td>E(q)(^{270})</td>
<td>—</td>
<td>Creates an engine with an initial state that depends on a sequence produced by one call to q.generate.</td>
<td>same as complexity of q.generate called on a sequence whose length is size of state</td>
</tr>
</tbody>
</table>

| e.seed()     | void        | post: e == E().                                                                  | same as E()         |
| e.seed(s)    | void        | post: e == E(s).                                                                 | same as E(s)        |
| e.seed(q)    | void        | post: e == E(q).                                                                 | same as E(q)        |
| e()          | T           | Advances e’s state \(e_i\) to \(e_{i+1}\) = \(TA(e_i)\) and returns \(GA(e_i)\). | per Table 110       |
| e.discard(z) | void        | Advances e’s state \(e_i\) to \(e_{i+z}\) by any means equivalent to \(z\) consecutive calls e(). | no worse than the complexity of \(z\) consecutive calls e() |
| x == y       | bool        | This operator is an equivalence relation. With \(S_x\) and \(S_y\) as the infinite sequences of values that would be generated by repeated future calls to \(x()\) and \(y()\), respectively, returns true if \(S_x = S_y\); else returns false. | \(O\) (size of state) |
| x != y       | bool        | \(! (x == y)\).                                                                | \(O\) (size of state) |
| os << x      | reference to the type of os | With os.fmtflags set to ios_base::dec|ios_base::left and the fill character set to the space character, writes to os the textual representation of x’s current state. In the output, adjacent numbers are separated by one or more space characters. post: The os.fmtflags and fill character are unchanged. | \(O\) (size of state) |

270) This constructor (as well as the subsequent corresponding seed() function) may be particularly useful to applications requiring a large number of independent random sequences.

271) This operation is common in user code, and can often be implemented in an engine-specific manner so as to provide significant performance improvements over an equivalent naive loop that makes \(z\) consecutive calls e().
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is &gt;&gt; v</code></td>
<td>reference to the type of <code>is</code></td>
<td>With <code>is.fmtflags</code> set to <code>ios_base::dec</code>, sets <code>v</code>'s state as determined by reading its textual representation from <code>is</code>. If bad input is encountered, ensures that <code>v</code>'s state is unchanged by the operation and calls <code>is.setstate(ios::failbit)</code> (which may throw <code>ios::failure [27.5.4.3]</code>). If a textual representation written via <code>os &lt;&lt; x</code> was subsequently read via <code>is &gt;&gt; v</code>, then <code>x == v</code> provided that there have been no intervening invocations of <code>x</code> or of <code>v</code>. pre: <code>is</code> provides a textual representation that was previously written using an output stream whose imbued locale was the same as that of <code>is</code>, and whose type’s template specialization arguments <code>charT</code> and <code>traits</code> were respectively the same as those of <code>is</code>. post: The <code>is.fmtflags</code> are unchanged.</td>
<td>$O(\text{size of state})$</td>
</tr>
</tbody>
</table>

30 E shall meet the requirements of CopyConstructible (Table 34) and CopyAssignable (Table 36) types. These operations shall each be of complexity no worse than $O(\text{size of state})$.

### 26.5.1.5 Concept RandomNumberEngineAdaptor

#### Random number engine adaptor requirements

A random number engine adaptor (commonly shortened to adaptor) a of type A is a random number engine that takes values produced by some other random number engine or engines, and applies an algorithm to those values in order to deliver a sequence of values with different randomness properties. An engine b of type B adapted in this way is termed a base engines in this context. The terms unary, binary, and so on, may be used to characterize an adaptor depending on the number n of base engines that adaptor utilizes.

The base engines of A are arranged in an arbitrary but fixed order, and that order is consistently used whenever functions are applied to those base engines in turn. In this context, the notation $b_i$ denotes the i\textsuperscript{th} of A’s base engines, $1 \leq i \leq n$, and $B_i$ denotes the type of $b_i$.

```cpp
concept RandomNumberEngineAdaptor<typename A, typename B0, typename... Bi> : RandomNumberEngine<A> {
  requires RandomNumberEngine<B0>
    && RandomNumberEngine<Bi>...
    && Constructible<A, const B0&, const Bi&...>
    && Constructible<A, B0&&, Bi&&...>;
}
```

§ 26.5.1.5 Numerics library 840
The requirements of a random number engine type shall be interpreted as follows with respect to a random number engine adaptor type.

```cpp
A::A(); // from RandomNumberEngine<A>
```

**Effects:** Each b. The base engine is initialized, in turn, as if by its respective default constructor.

```cpp
bool operator==(const A& a1, const A& a2); // from RandomNumberEngine<A>
```

**Returns:** true if each pair of corresponding b. a1’s base engine is equal to a2’s base engine are equal. Otherwise returns false.

```cpp
A::A(result_type s); // from RandomNumberEngine<A>
```

**Effects:** Each b. The base engine is initialized, in turn, with the next available value from the list s + 0, s + 1, ...

```cpp
template<class Sseq> void A::A(Sseq& q); // from RandomNumberEngine<A>
```

**Effects:** Each b. The base engine is initialized, in turn, with q as argument.

```cpp
void seed(A& a); // from RandomNumberEngine<A>
```

**Effects:** For each b, in turn, invokes RandomNumberEngine::seed(b) - With b as the base engine, invokes b.seed().

```cpp
void seed(A& a, result_type s); // from RandomNumberEngine<A>
```

**Effects:** For each b, in turn, invokes RandomNumberEngine::seed(b, s) with the next available value from the list s + 0, s + 1, ... - With b as the base engine, invokes b.seed(s).

```cpp
template<class Sseq> void seed(A& a, Sseq& q); // from RandomNumberEngine<A>
```

**Effects:** For each b, in turn, invokes RandomNumberEngine::seed(b, q) - With b as the base engine, invokes b.seed(q).

A shall also satisfy the following additional requirements:

a) The complexity of each function shall be at most the sum of the complexities of the corresponding functions applied to each not exceed the complexity of the corresponding function applied to the base engine.

b) The state of A shall include the state of each of its base engines. The size of A’s state shall be no less than the sum of the base engines’ respective sizes.

c) Copying A’s state (e.g., during copy construction or copy assignment) shall include copying, in turn, the state of each base engine of A.

d) The textual representation of A shall include, in turn, the textual representation of each of its base engines.

e) Any constructor satisfying the requirement Constructible<A, const RandomNumberEngine&...> or satisfying the requirement Constructible<A, RandomNumberEngine&&...> shall have n or more parameters such that the underlying type of parameter i, 1 ≤ i ≤ n, is B_i, and such that all remaining parameters, if any, have default values. The constructor shall create an engine adaptor initializing each b, in turn, with a copy of the value of the corresponding argument.

§ 26.5.1.5 Numerics library

841
26.5.1.6 Concept RandomNumberDistribution

Random number distribution requirements

**[rand.req.dist]**

1 A *random number distribution* (commonly shortened to *distribution*) \(d\) of type \(D\) is a function object returning values that are distributed according to an associated mathematical *probability density function* \(p(z)\) or according to an associated *discrete probability function* \(P(z_i)\). A distribution’s specification identifies its associated probability function \(p(z)\) or \(P(z_i)\).

2 An associated probability function is typically expressed using certain externally-supplied quantities known as the *parameters* of the distribution. Such distribution parameters are identified in this context by writing, for example, \(p(z|a,b)\) or \(P(z_i|a,b)\), to name specific parameters, or by writing, for example, \(p(z|\{p\})\) or \(P(z_i|\{p\})\), to denote a distribution’s parameters \(p\) taken as a whole.

```cpp
concept RandomNumberDistribution<typename D> : Regular<D> {
    ArithmeticType result_type;
    Regular param_type;
    requires Constructible<D, const param_type&>;

    void reset(D& d);

    template<UniformRandomNumberGenerator URNG> result_type operator()(D& d, URNG& g);
    template<UniformRandomNumberGenerator URNG> result_type operator()(D& d, URNG& g, const param_type& p);

    param_type param(const D& d);
    void param(D& d, const param_type&);

    result_type min(const D& d);
    result_type max(const D& d);

    template<OutputStreamable OS> OS& operator<<(OS& os, const D& d);
    template<InputStreamable IS> IS& operator>>(IS& is, D& d);
}
```

Regular param_type;

3 **Requires:**

   a) For each of the constructors of \(D\) taking arguments corresponding to parameters of the distribution, \param_type\ shall provide a corresponding constructor subject to the same requirements and taking arguments identical in number, type, and default values.

   b) For each of the member functions of \(D\) that return values corresponding to parameters of the distribution, \param_type\ shall provide a corresponding member function with the identical name, type, and semantics.

   c) \param_type\ shall provide a declaration of the form `typedef D distribution_type;`.

4 **Remark:** It is unspecified whether \param_type\ is declared as a (nested) *class* or via a *typedef*. In this subclause 26.5, declarations of \(D::\param_type\) are in the form of *typedefs* only for convenience of exposition.

\(D::D(const\ param_type& p);\)

5 **Effects:** Creates a distribution whose behavior is indistinguishable from that of a distribution newly created directly from the values used to create \(p\).

6 **Complexity:** same as \(p\)’s construction.

```cpp
bool operator==(const D& d1, const D& d2); // from Regular<D>
```

7 **Returns:** `true` if \(d1.param() == d2.param()\) and \(S_1 = S_2\), where \(S_1\) and \(S_2\) are the infinite sequences of values that would be generated, respectively, by repeated future calls to \(d1(g1)\) and \(d2(g2)\) whenever \(g1 == g2\). Otherwise returns `false`.

§ 26.5.1.6 Numerics library
void reset(D& d);

Effects: Subsequent uses of the distribution do not depend on values produced by any engine prior to invoking reset.
Complexity: constant.

\[
\text{template<UniformRandomNumberGenerator URNG> result_type operator() (D& d, URNG& g);} \\
\]

Effects: With \( p = \text{param()} \), the sequence of numbers returned by successive invocations with the same object \( u \) is randomly distributed according to the associated probability function \( p(z \mid \{p\}) \) or \( P(z_i \mid \{p\}) \).
For distributions \( x \) and \( y \) of identical type \( D \):

a) The sequence of numbers produced by repeated invocations of \( x(u) \) shall be independent of any invocation of \( \text{os} \ll x \) or of any \( \text{const} \) member function of \( D \) between any of the invocations \( x(u) \).
b) If a textual representation is written using \( \text{os} \ll x \) and that representation is restored into the same or a different object \( y \) using \( \text{is} \gg y \), repeated invocations of \( y(u) \) shall produce the same sequence of numbers as would repeated invocations of \( x(u) \).
Complexity: amortized constant number of invocations of \( u \).

\[
\text{template<UniformRandomNumberGenerator URNG> result_type operator() (D& d, URNG& g, const param_type& p);} \\
\]

Effects: The sequence of numbers returned by successive invocations with the same objects \( g \) and \( p \) is randomly distributed according to the associated probability function \( p(z \mid \{p\}) \) or \( P(z_i \mid \{p\}) \).

\[
\text{param_type param(const D& d);} \\
\]

Returns: a value \( p \) such that \( \text{param}(D(p)) == p \).
Complexity: no worse than the complexity of \( D(p) \).

\[
\text{void param(D& d, const param_type& p);} \\
\]

Postcondition: \( \text{param}(d) == p \).
Complexity: no worse than the complexity of \( D(p) \).

\[
\text{result_type min(const D& d);} \\
\]

Returns: the greatest lower bound on the values potentially returned by \( \text{operator()} \), as determined by the current values of the distribution's parameters.
Complexity: constant.

\[
\text{result_type max(const D& d);} \\
\]

Returns: the least upper bound on the values potentially returned by \( \text{operator()} \), as determined by the current values of the distribution's parameters.
Complexity: constant.

\[
\text{template<OutputStreamable OS> OS& operator<<(OS& os, const D& d);} \\
\]

Effects: Writes to \( \text{os} \) a textual representation for the parameters and the additional internal data of \( d \).
Returns: the updated \( \text{os} \).
Postcondition: The \( \text{os}.\text{fmtflags} \) and fill character are unchanged.

\[
\text{template<InputStreamable IS> IS& operator>>(IS& is, D& d);} \\
\]
24 Precondition: is provides a textual representation that was previously written using an output stream whose imbued locale was the same as that of is, and whose associated types OutputStreamable::charT and OutputStreamable::traits were respectively the same as those of is.

25 Effects: Restores from is the parameters and the additional internal data of d. If bad input is encountered, ensures that d is unchanged by the operation and calls is.setstate(ios::failbit) (which may throw ios::failure [27.5.4.3]).

26 Returns: the updated is.

27 Postcondition: The is.fmtflags are unchanged.

28 A class D satisfies the requirements of a random number distribution if the expressions shown in Table 112 are valid and have the indicated semantics, and if D and its associated types also satisfy all other requirements of this section 26.5.1.6. In that Table and throughout this section,

a) T is the type named by D’s associated result_type;

b) P is the type named by D’s associated param_type;

c) d is a value of D, and x and y are (possibly const) values of D;

d) glb and lub are values of T respectively corresponding to the greatest lower bound and the least upper bound on the values potentially returned by d’s operator(), as determined by the current values of d’s parameters;

e) p is a (possibly const) value of P;

f) g, g1, and g2 are lvalues of a type satisfying the requirements of a uniform random number generator [26.5.1.3];

g) os is an lvalue of the type of some class template specialization basic_ostream<charT, traits>; and

h) is is an lvalue of the type of some class template specialization basic_istream<charT, traits>; where charT and traits are constrained according to Clause 21 and 27.

Table 112 — Random number distribution requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>D::result_type</td>
<td>T</td>
<td>T is an arithmetic type (3.9.1).</td>
<td>compile-time</td>
</tr>
<tr>
<td>D::param_type</td>
<td>P</td>
<td>—</td>
<td>compile-time</td>
</tr>
<tr>
<td>D()</td>
<td>—</td>
<td>Creates a distribution whose behavior is indistinguishable from that of any other newly default-constructed distribution of type D.</td>
<td>constant</td>
</tr>
<tr>
<td>D(p)</td>
<td>—</td>
<td>Creates a distribution whose behavior is indistinguishable from that of a distribution newly constructed directly from the values used to construct p.</td>
<td>same as p’s construction</td>
</tr>
<tr>
<td>d.reset()</td>
<td>void</td>
<td>Subsequent uses of d do not depend on values produced by any engine prior to invoking reset.</td>
<td>constant</td>
</tr>
<tr>
<td>x.param()</td>
<td>P</td>
<td>Returns a value p such that D(p).param() == p.</td>
<td>no worse than the complexity of D(p)</td>
</tr>
<tr>
<td>d.param(p)</td>
<td>void</td>
<td>post: d.param() == p.</td>
<td>no worse than the complexity of D(p)</td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Pre/post-condition</td>
<td>Complexity</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>--------------------</td>
<td>------------</td>
</tr>
<tr>
<td>d(g)</td>
<td>T</td>
<td>With ( p = d.\text{param}() ), the sequence of numbers returned by successive invocations with the same object ( g ) is randomly distributed according to the associated ( p(z</td>
<td>{p}) ) or ( P(z_i</td>
</tr>
<tr>
<td>d(g,p)</td>
<td>T</td>
<td>The sequence of numbers returned by successive invocations with the same objects ( g ) and ( p ) is randomly distributed according to the associated ( p(z</td>
<td>{p}) ) or ( P(z_i</td>
</tr>
<tr>
<td>x.min()</td>
<td>T</td>
<td>Returns ( \text{glb} ).</td>
<td>constant</td>
</tr>
<tr>
<td>x.max()</td>
<td>T</td>
<td>Returns ( \text{lub} ).</td>
<td>constant</td>
</tr>
<tr>
<td>x == y</td>
<td>bool</td>
<td>This operator is an equivalence relation. Returns ( \text{true} ) if ( x.\text{param}() == y.\text{param}() ) and ( S_1 = S_2 ), where ( S_1 ) and ( S_2 ) are the infinite sequences of values that would be generated, respectively, by repeated future calls to ( x(g_1) ) and ( y(g_2) ) whenever ( g_1 == g_2 ). Otherwise returns ( \text{false} ).</td>
<td>constant</td>
</tr>
<tr>
<td>x != y</td>
<td>bool</td>
<td>(! (x == y)).</td>
<td>same as ( x == y )</td>
</tr>
<tr>
<td>os &lt;&lt; x</td>
<td>reference to the type of ( os )</td>
<td>Writes to ( os ) a textual representation for the parameters and the additional internal data of ( x ). post: The ( os.\text{fmtflags} ) and fill character are unchanged.</td>
<td>—</td>
</tr>
</tbody>
</table>
Expression | Return type | Pre/post-condition | Complexity |
---|---|---|---|
`is >> d` | reference to the type of `is` | Restores from `is` the parameters and additional internal data of the lvalue `d`. If bad input is encountered, ensures that `d` is unchanged by the operation and calls `is.setstate(ios::failbit)` (which may throw `ios::failure [27.5.4.3]`). | — |

29 D shall satisfy the requirements of CopyConstructible (Table 34) and CopyAssignable (Table 36) types.
30 The sequence of numbers produced by repeated invocations of `d(g)` shall be independent of any invocation of `os << d` or of any `const` member function of D between any of the invocations `d(g)`.
31 If a textual representation is written using `os << x` and that representation is restored into the same or a different object `y` of the same type using `is >> y`, repeated invocations of `y(g)` shall produce the same sequence of numbers as would repeated invocations of `x(g)`.
32 It is unspecified whether D::param_type is declared as a (nested) class or via a typedef. In this subclause 26.5, declarations of D::param_type are in the form of typedefs for convenience of exposition only.
33 P shall satisfy the requirements of CopyConstructible (Table 34), CopyAssignable (Table 36), and Equality-Comparable (Table 31) types.
34 For each of the constructors of D taking arguments corresponding to parameters of the distribution, P shall have a corresponding constructor subject to the same requirements and taking arguments identical in number, type, and default values. Moreover, for each of the member functions of D that return values corresponding to parameters of the distribution, P shall have a corresponding member function with the identical name, type, and semantics.
35 P shall have a declaration of the form

```cpp
typedef D distribution_type;
```

26.5.2 Header `<random>` synopsis

[Editor’s note: This synopsis section has been relocated here; it formerly preceded the Concepts (now Requirements) section above.]

```cpp
namespace std {
    #include <initializer_list>

    // 26.5.1.2 Concept SeedSequence
    concept SeedSequence<typename S>

    // 26.5.1.3 Concept UniformRandomNumberGenerator
    concept UniformRandomNumberGenerator<typename UD>
```

§ 26.5.2 Numerics library 846
// 26.5.1.4 Concept RandomNumberEngine
concept RandomNumberEngine<typename E>

// 26.5.1.5 Concept RandomNumberEngineAdaptor
concept RandomNumberEngineAdaptor<typename A>

// 26.5.1.6 Concept RandomNumberDistribution
concept RandomNumberDistribution<typename D>

// 26.5.3.1 Class template linear_congruential_engine
template<UnsignedIntegralLike class UIntType, UIntType a, UIntType c, UIntType m>
requires IntegralType<UIntType>
&& True<m == 0u || (a < m && c < m)>
class linear_congruential_engine;

// 26.5.3.2 Class template mersenne_twister_engine
template<UnsignedIntegralLike class UIntType, size_t w, size_t n, size_t m, size_t r,
UIntType a, size_t u, UIntType d, size_t s,
UIntType b, size_t t, UIntType c, size_t l, UintType f>
requires IntegralType<UIntType>
&& True<s < m && m < n
&& r < w && u <= w && g <= u && t <= u && 1 < u
&& u <= numeric_limits<UIntType>::digits
&& a < (1u<<w) - 1u && b <= (1u<<w) - 1u && c <= (1u<<w) - 1u>
class mersenne_twister_engine;

// 26.5.3.3 Class template subtract_with_carry_engine
template<UnsignedIntegralLike class UIntType, size_t w, size_t s, size_t r>
requires IntegralType<UIntType>
&& True<s < w && 0 < w && w <= numeric_limits<UIntType>::digits>
class subtract_with_carry_engine;

// 26.5.4.1 Class template discard_block_engine
template<RandomNumberEngine<typename Engine, size_t p, size_t r>
requires True<p == r && r == p>
class discard_block_engine;

// 26.5.4.2 Class template independent_bits_engine
template<RandomNumberEngine<typename Engine, size_t w, UnsignedIntegralLike class UIntType>
requires IntegralType<UIntType>
&& True<w && w <= numeric_limits<result_type>::digits>
class independent_bits_engine;

// 26.5.4.3 Class template shuffle_order_engine
template<RandomNumberEngine<typename Engine, size_t k>
requires True<k <= k>
class shuffle_order_engine;

// 26.5.5 Engines and engine adaptors with predefined parameters
typedef see below minstd_rand0;
typedef see below minstd_rand;
typedef see below mt19937;
typedef see below mt19937_64;
typedef see below ranlux24_base;

§ 26.5.2 Numerics library 847
typedef see below ranlux48_base;
typedef see below ranlux24;
typedef see below ranlux48;
typedef see below knuth_b;
typedef see below default_random_engine;

// 26.5.6 Class random_device
class random_device;

// 26.5.7.1 Class seed_seq
class seed_seq;

// 26.5.7.2 Function template generate_canonical
template<FloatingPointLike class RealType, size_t bits, UniformRandomNumberGenerator class URNG>
    requires FloatingPointType<RealType>
    RealType generate_canonical(URNG& g);

// 26.5.8.1.1 Class template uniform_int_distribution
template<IntegralLike class IntType = int>
    requires IntegralType<IntType>
    class uniform_int_distribution;

// 26.5.8.1.2 Class template uniform_real_distribution
template<FloatingPointLike class RealType = double>
    requires FloatingPointType<RealType>
    class uniform_real_distribution;

// 26.5.8.2.1 Class bernoulli_distribution
class bernoulli_distribution;

// 26.5.8.2.2 Class template binomial_distribution
template<IntegralLike class IntType = int>
    requires IntegralType<IntType>
    class binomial_distribution;

// 26.5.8.2.3 Class template geometric_distribution
template<IntegralLike class IntType = int>
    requires IntegralType<IntType>
    class geometric_distribution;

// 26.5.8.2.4 Class template negative_binomial_distribution
template<IntegralLike class IntType = int>
    requires IntegralType<IntType>
    class negative_binomial_distribution;

// 26.5.8.3.1 Class template poisson_distribution
template<IntegralLike class IntType = int>
    requires IntegralType<IntType>
    class poisson_distribution;

// 26.5.8.3.2 Class template exponential_distribution
template<FloatingPointLike class RealType = double>
    requires FloatingPointType<RealType>
    class exponential_distribution;
// 26.5.8.3.3 Class template gamma_distribution
template<
    FloatingPointLike class RealType = double>
    requires FloatingPointType<RealType>
    class gamma_distribution;

// 26.5.8.3.4 Class template weibull_distribution
template<
    FloatingPointLike class RealType = double>
    requires FloatingPointType<RealType>
    class weibull_distribution;

// 26.5.8.3.5 Class template extreme_value_distribution
template<
    FloatingPointLike class RealType = double>
    requires FloatingPointType<RealType>
    class extreme_value_distribution;

// 26.5.8.4.1 Class template normal_distribution
template<
    FloatingPointLike class RealType = double>
    requires FloatingPointType<RealType>
    class normal_distribution;

// 26.5.8.4.2 Class template lognormal_distribution
template<
    FloatingPointLike class RealType = double>
    requires FloatingPointType<RealType>
    class lognormal_distribution;

// 26.5.8.4.3 Class template chi_squared_distribution
template<
    FloatingPointLike class RealType = double>
    requires FloatingPointType<RealType>
    class chi_squared_distribution;

// 26.5.8.4.4 Class template cauchy_distribution
template<
    FloatingPointLike class RealType = double>
    requires FloatingPointType<RealType>
    class cauchy_distribution;

// 26.5.8.4.5 Class template fisher_f_distribution
template<
    FloatingPointLike class RealType = double>
    requires FloatingPointType<RealType>
    class fisher_f_distribution;

// 26.5.8.4.6 Class template student_t_distribution
template<
    FloatingPointLike class RealType = double>
    requires FloatingPointType<RealType>
    class student_t_distribution;

// 26.5.8.5.1 Class template discrete_distribution
template<
    IntegralLike class IntType = int>
    requires IntegralType<IntType>
    class discrete_distribution;

// 26.5.8.5.2 Class template piecewise_constant_distribution
template<
    FloatingPointLike class RealType = double>
    requires FloatingPointType<RealType>
    class piecewise_constant_distribution;
26.5.3 Random number engine class templates

Each type instantiated from a class template specified in this section 26.5.3 satisfies the requirements of a random number engine (26.5.1.4) type. Except where specified otherwise, the complexity of all each function specified in the following sections this section 26.5.3 is constant. Except where specified otherwise, no function described in this section 26.5.3 throws an exception. For every class E instantiated from a template specified in this section 26.5.3, a concept map RandomNumberEngine<E> shall be defined in namespace std so as to provide mappings from free functions to the corresponding member functions. Descriptions are provided herein this section 26.5.3 only for engine operations that are not described in 26.5.1.4 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

Each template specified in this section 26.5.3 requires one or more relationships, involving the value(s) of its non-type template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

26.5.3.1 Class template linear_congruential_engine

A linear_congruential_engine random number engine produces unsigned integer random numbers. The state x_i of a linear_congruential_engine object x is of size 1 and consists of a single integer. The transition algorithm is a modular linear function of the form TA(x_i) = (a · x_i + c) mod m; the generation algorithm is GA(x_i) = x_i + 1.
If the template parameter m is 0, the modulus m used throughout this section 26.5.3.1 is \texttt{numeric_limits<result_type>::max()} plus 1. [Note: m need not be representable as a value of type \texttt{result_type}. —end note]

If the template parameter m is not 0, the following relations shall hold: \( a < m \) and \( c < m \).

The textual representation consists of the value of \( x_i \).

\begin{verbatim}
explicit linear_congruential_engine(result_type s = default_seed);
\end{verbatim}

\textbf{Effects:} Constructs a \texttt{linear_congruential_engine} object. If \( c \mod m \) is 0 and \( s \mod m \) is 0, sets the engine’s state to 1, otherwise sets the engine’s state to \( s \mod m \).

\begin{verbatim}
template<SeedSequence class Sseq> explicit linear_congruential_engine(Sseq& q);
\end{verbatim}

\textbf{Effects:} Constructs a \texttt{linear_congruential_engine} object. With \( k = \left\lceil \frac{\log_2 m}{32} \right\rceil \) and \( a \) an array (or equivalent) of length \( k + 3 \), invokes \texttt{q.generate(q, a + 0, a + k + 3)} and then computes \( S = \left( \sum_{j=0}^{k-1} a_j + 3 \cdot 2^{32} \right) \mod m \). If \( c \mod m \) is 0 and \( S \) is 0, sets the engine’s state to 1, else sets the engine’s state to \( S \).

26.5.3.2 Class template \texttt{mersenne_twister_engine}

A \texttt{mersenne_twister_engine} random number engine\textsuperscript{272} produces unsigned integer random numbers in the closed interval \([0, 2^w - 1]\). The state \( x_i \) of a \texttt{mersenne_twister_engine} object \( x \) is of size \( n \) and consists of a sequence \( X \) of \( n \) values of the type delivered by \( x \); all subscripts applied to \( X \) are to be taken modulo \( n \).

The transition algorithm employs a twisted generalized feedback shift register defined by shift values \( u \) and \( m \), a twist value \( r \), and a conditional xor-mask \( a \). To improve the uniformity of the result, the bits of the raw shift register are additionally \textit{tempered} (\textit{i.e.}, scrambled) according to a bit-scrambling matrix defined by values \( u, d, s, b, t, c, \) and \( \ell \).

The state transition is performed as follows:

a) Concatenate the upper \( w - r \) bits of \( X_{i-n} \) with the lower \( r \) bits of \( X_{i+1-n} \) to obtain an unsigned integer value \( Y \).

b) With \( a = a \cdot (Y \texttt{bitand} 1) \), set \( X_i \) to \( X_{i+m-n} \texttt{xor} (Y \texttt{rshift} 1) \texttt{xor} a \).

The sequence \( X \) is initialized with the help of an initialization multiplier \( f \).

3 The generation algorithm determines the unsigned integer values \( z_1, z_2, z_3, z_4 \) as follows, then delivers \( z_4 \) as its result:

a) Let \( z_1 = X_i \texttt{xor} ((X_i \texttt{rshift} u) \texttt{bitand} d) \).

b) Let \( z_2 = z_1 \texttt{xor} ((z_1 \texttt{lshift} s) \texttt{bitand} b) \).

c) Let \( z_3 = z_2 \texttt{xor} ((z_2 \texttt{lshift} t) \texttt{bitand} c) \).

d) Let \( z_4 = z_3 \texttt{xor} (z_3 \texttt{rshift} \ell) \).

\begin{verbatim}
template<UnsignedIntegralLike class UINTType, size_t w, size_t n, size_t m, size_t r,
    UINTType a, size_t u, UINTType d, size_t s,
    UINTType b, size_t t, UINTType c,
    size_t l, UINTType f>
requires IntegralTypes<UINTType>
& & True<\_u <= w && m <= n
& & \_a <= 1u && b <= 1u && c <= 1u && d <= 1u && f <= 1u
& & a <= (1uw) - 1u && b <= (1uw) - 1u && c <= (1uw) - 1u
& & d <= (1uw) - 1u && f <= (1uw) - 1u
class mersenne_twister_engine
\end{verbatim}

\textsuperscript{272} The name of this engine refers, in part, to a property of its period: For properly-selected values of the parameters, the period is closely related to a large Mersenne prime number.

\[\text{§ 26.5.3.2 Numerics library} \quad 851\]
```cpp
{  
public:  
  // types  
typedef UIntType result_type;
  
  // engine characteristics  
  static constexpr size_t word_size = w;  
  static constexpr size_t state_size = n;  
  static constexpr size_t shift_size = m;  
  static constexpr size_t mask_bits = r;  
  static constexpr UIntType xor_mask = a;  
  static constexpr size_t tempering_u = u;  
  static constexpr UIntType tempering_d = d;  
  static constexpr size_t tempering_s = s;  
  static constexpr UIntType tempering_t = t;  
  static constexpr size_t tempering_l = l;  
  static constexpr UIntType initialization_multiplier = f;
  static constexpr result_type min() { return 0; }  
  static constexpr result_type max() { return 2w - 1; }  
  static constexpr result_type default_seed = 5489u;
  
  // constructors and seeding functions  
  explicit mersenne_twister_engine(result_type value = default_seed);
  template<SeedSequence class Sseq> explicit mersenne_twister_engine(Sseq& q);
  void seed(result_type value = default_seed);
  template<SeedSequence class Sseq> void seed(Sseq& q);
  
  // generating functions  
  result_type operator()();  
  void discard(unsigned long long z);
};

4 The following relations shall hold: 0 < m, m <= n, r <= w, u <= w, s <= w, t <= w, l <= w, w <= numeric_limits<UIntType>::digits, a <= (1u«w) - 1u, b <= (1u«w) - 1u, c <= (1u«w) - 1u, and f <= (1u«w) - 1u.

5 The textual representation of \( x_i \) consists of the values of \( X_{i-n}, \ldots, X_{i-1}, \) in that order.

6 **Effects:** Constructs a **mersenne_twister_engine** object. Sets \( X_{-n} \) to value mod \( 2^w \). Then, iteratively for \( i = 1-n, \ldots, -1 \), sets \( X_i \) to

\[
[f \cdot (X_{i-1} \xor (X_{i-1} \text{rshift}(w - 2))) + i \text{ mod } n] \mod 2^w.
\]

7 **Complexity:** \( \Theta(n) \).

8 **Effects:** Constructs a **mersenne_twister_engine** object. With \( k = \lceil w/32 \rceil \) and \( a \) an array (or equivalent) of length \( n \cdot k \), invokes \( \text{generate}(\cdot, a + 0, a + n \cdot k) \) and then, iteratively for \( i = -n, \ldots, -1 \), sets \( X_i \) to \( (\sum_{j=0}^{k-1} a_{k(i+n)+j} \cdot 2^{32j}) \mod 2^w \). Finally, if the most significant \( w - r \) bits of \( X_{-n} \) are zero, and if each of the other resulting \( X_i \) is 0, changes \( X_{-n} \) to \( 2^w - 1 \).

§ 26.5.3.2 Numerics library

852
26.5.3.3 Class template subtract_with_carry_engine

A `subtract_with_carry_engine` random number engine produces unsigned integer random numbers.

The state $x_i$ of a `subtract_with_carry_engine` object $x$ is of size $O(r)$, and consists of a sequence $X$ of $r$ integer values $0 \leq X_i < m = 2^w$; all subscripts applied to $X$ are to be taken modulo $r$. The state $x_i$ additionally consists of an integer $c$ (known as the carry) whose value is either 0 or 1.

The state transition is performed as follows:

a) Let $Y = X_{i-s} - X_{i-r} - c$.

b) Set $X_i$ to $y = Y \mod m$. Set $c$ to 1 if $Y < 0$, otherwise set $c$ to 0.

[Note: This algorithm corresponds to a modular linear function of the form $T_A(x_i) = (a \cdot x_i) \mod b$, where $b$ is of the form $m^r - m^s + 1$ and $a = b - (b - 1)/m$. — end note]

The generation algorithm is given by $G_A(x_i) = y$, where $y$ is the value produced as a result of advancing the engine’s state as described above.

```cpp
template<UnsignedIntegralLike class UIntType, size_t w, size_t s, size_t r>
requires IntegralType<UIntType> && True<0u < s && s < r && 0 < w && w <= numeric_limits<UIntType>::digits>
class subtract_with_carry_engine
{
  public:
    // types
    typedef UIntType result_type;

    // engine characteristics
    static constexpr size_t word_size = w;
    static constexpr size_t short_lag = s;
    static constexpr size_t long_lag = r;
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return m - 1; }
    static constexpr result_type default_seed = 19780503u;

    // constructors and seeding functions
    explicit subtract_with_carry_engine(result_type value = default_seed);
    template<Class Sseq> explicit subtract_with_carry_engine(Sseq& q);
    void seed(result_type value = default_seed);
    template<Class Sseq> void seed(Sseq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
};
```

The following relations shall hold: $0u < s, s < r, 0 < w$, and $w \leq \text{numeric_limits}<\text{UIntType}>::\text{digits}$.

The textual representation consists of the values of $X_{i-r}, \ldots, X_{i-1}$, in that order, followed by $c$.

```cpp
explicit subtract_with_carry_engine(result_type value = default_seed);
```

Effects: Constructs a `subtract_with_carry_engine` object. Sets the values of $X_{i-r}, \ldots, X_{i-1}$, in that order, as specified below. If $X_{i-1}$ is then 0, sets $c$ to 1; otherwise sets $c$ to 0.

To set the values $X_k$, first construct $e$, a `linear_congruential_engine` object, as if by the following definition:

```cpp
linear_congruential_engine<result_type,
  40014u, 0u, 2147483563u> e(value == 0u ? default_seed : value);
```
Then, to set each $X_k$, obtain new values $z_0, \ldots, z_{n-1}$ from $n = \lceil w/32 \rceil$ successive invocations of $e$ taken modulo $2^{32}$. Set $X_k$ to $(\sum_{j=0}^{n-1} z_j \cdot 2^{32j}) \mod m$.

**Complexity:** Exactly $n \cdot r$ invocations of $e$.

```cpp
template<SeedSequence class Sseq> explicit subtract_with_carry_engine(Sseq& q);
```

**Effects:** Constructs a `subtract_with_carry_engine` object. With $k = \lceil w/32 \rceil$ and $a$ an array (or equivalent) of length $r \cdot k$, invokes $q$ generate($q$, $a$ + 0, $a$ + $r \cdot k$) and then, iteratively for $i = -r, \ldots, -1$, sets $X_i$ to $(\sum_{j=0}^{k-1} a_{k(i+r)+j} \cdot 2^{32j}) \mod m$. If $X_{-1}$ is then 0, sets $c$ to 1; otherwise sets $c$ to 0.

### 26.5.4 Random number engine adaptor class templates

1. Each type instantiated from a class template specified in this section 26.5.3 satisfies the requirements of a random number engine adaptor (26.5.1.5) type.
2. Except where specified otherwise, the complexity of all each functions specified in the following sections this section 26.5.4 is constant.
3. Except where specified otherwise, no function described in this section 26.5.4 throws an exception.
4. For every class A instantiated from a template specified in this section 26.5.4, a concept map RandomNumberEngineAdaptor<E> shall be defined in namespace std so as to provide mappings from free functions to the corresponding member functions. Descriptions are provided herein this section 26.5.4 only for adaptor operations that are not described in section 26.5.1.5 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.
5. Each template specified in this section 26.5.4 requires one or more relationships, involving the value(s) of its non-type template parameter(s), to hold. A program instantiating any of these templates is ill-formed if any such required relationship fails to hold.

#### 26.5.4.1 Class template discard_block_engine

1. A `discard_block_engine` random number engine adaptor produces random numbers selected from those produced by some base engine $e$. The state $x_i$ of a `discard_block_engine` engine adaptor object $x$ consists of the state $e_i$ of its base engine $e$ and an additional integer $n$. The size of the state is the size of $e$’s state plus 1.
2. The transition algorithm discards all but $r > 0$ values from each block of $p \geq r$ values delivered by $e$. The state transition is performed as follows: If $n \geq r$, advance the state of $e$ from $e_i$ to $e_i + p - r$ and set $n$ to 0. In any case, then increment $n$ and advance $e$’s then-current state $e_j$ to $e_j + 1$.
3. The generation algorithm yields the value returned by the last invocation of $e()$ while advancing $e$’s state as described above.

```cpp
template<typename Engine, size_t p, size_t r>
requires True<\l r && r \leq p>
class discard_block_engine
{
    public:
        // types
        typedef typename Engine::result_type result_type;

        // engine characteristics
        static constexpr size_t block_size = p;
        static constexpr size_t used_block = r;
        static constexpr result_type min() { return Engine::min; }
        static constexpr result_type max() { return Engine::max; }

        // constructors and seeding functions
```
discard_block_engine();
explicit discard_block_engine(const Engine& e);
explicit discard_block_engine(Engine&& e);
explicit discard_block_engine(result_type s);
template<SeedSequence class Sseq>
explicit discard_block_engine(Sseq& q);
void seed();
void seed(result_type s);
template<SeedSequence class Sseq>
void seed(Sseq& q);

// generating functions
result_type operator()();
void discard(unsigned long long z);

// property functions
const Engine& base() const;
private:
Engine e; // exposition only
int n; // exposition only
};

4 The following relations shall hold: 0 < r and r <= p.
5 The textual representation consists of the textual representation of e followed by the value of n.
6 In addition to its behavior pursuant to section 26.5.1.5, each constructor that is not a copy constructor sets n to 0.

26.5.4.2 Class template independent_bits_engine

An independent_bits_engine random number engine adaptor combines random numbers that are produced by some base engine e, so as to produce random numbers with a specified number of bits w. The state xᵢ of an independent_bits_engine engine adaptor object x consists of the state eᵢ of its base engine e; the size of the state is the size of e’s state.

2 The transition and generation algorithms are described in terms of the following integral constants:
   a) Let R = e.max() - e.min() + 1 and m = ⌊log₂ R⌋.
   b) With n as determined below, let w₀ = ⌊w/n⌋, n₀ = n - w mod n, y₀ = 2ʷ₀ ⌊R/2ʷ₀⌋, and y₁ = 2ʷ₀⁺¹ ⌊R/2ʷ₀⁺¹⌋.
   c) Let n = ⌊w/m⌋ if and only if the relation R - y₀ ≤ ⌊y₀/n⌋ holds as a result. Otherwise let n = 1 + ⌊w/m⌋.
   [Note: The relation w = n₀w₀ + (n - n₀)(w₁ + 1) always holds. —end note]
3 The transition algorithm is carried out by invoking e() as often as needed to obtain n₀ values less than y₀ + e.min() and n - n₀ values less than y₁ + e.min().
4 The generation algorithm uses the values produced while advancing the state as described above to yield a quantity S obtained as if by the following algorithm:

\[ S = 0; \]
\[ \text{for} \ (k = 0; \ k ≠ n₀; \ k ++) \ \{ \]
\[ \quad \text{do} \ u = e() - e.min(); \text{while} \ (u ≥ y₀); \]
\[ \quad S = 2^{w₀} \cdot S + u \mod 2^{w₀}; \]
\[ \} \]
\[ \text{for} \ (k = n₀; \ k ≠ n; \ k ++) \ \{ \]
\[ \quad \text{do} \ u = e() - e.min(); \text{while} \ (u ≥ y₁); \]
\[ \quad S = 2^{w₀⁺¹} \cdot S + u \mod 2^{w₀⁺¹}; \]
\[ \} \]

template<RandomNumberEngine class Engine, size_t w, UnsignedIntegralLike class UIntType>
requires IntegralType<UIntType>
& True<0u < w && w <= numeric_limits<result_type>::digits

class independent_bits_engine
{
    public:
        // types
typedef UIntType result_type;

        // engine characteristics
        static constexpr result_type min() { return 0; }
        static constexpr result_type max() { return 2^w - 1; }

        // constructors and seeding functions
        independent_bits_engine();
        explicit independent_bits_engine(const Engine& e);
        explicit independent_bits_engine(Engine&& e);
        explicit independent_bits_engine(result_type s);
        template<SeedSequence class Sseq> explicit independent_bits_engine(Sseq& q);
        void seed();
        void seed(result_type s);
        template<SeedSequence class Sseq> void seed(Sseq& q);

        // generating functions
        result_type operator()();
        void discard(unsigned long long z);

        // property functions
        const Engine& base() const;

    private:
        Engine e;   // exposition only
};

5 The following relations shall hold: 0 < w and w <= numeric_limits<result_type>::digits.

6 The textual representation consists of the textual representation of e.

26.5.4.3 Class template shuffle_order_engine [rand.adapt.shuf]

1 A shuffle_order_engine random number engine adaptor produces the same random numbers that are
produced by some base engine e, but delivers them in a different sequence. The state x_i of a shuffle_-
order_engine engine adaptor object x consists of the state e_i of its base engine e, an additional value Y of
the type delivered by e, and an additional sequence V of k values also of the type delivered by e. The size
of the state is the size of e's state plus k + 1.

2 The transition algorithm permutes the values produced by e. The state transition is performed as follows:
   a) Calculate an integer j = \left\lfloor \frac{k \cdot (Y - e_{\min})}{e_{\max} - e_{\min} + 1} \right\rfloor.
   b) Set Y to V_j and then set V_j to e().

3 The generation algorithm yields the last value of Y produced while advancing e's state as described above.

    template<RandomNumberEngine class Engine, size_t k>
    requires True<1u < k>
    class shuffle_order_engine
    {
        public:
            // types
typedef typename Engine::result_type result_type;

            // engine characteristics

§ 26.5.4.3 Numerics library 856
static constexpr size_t table_size = k;
static constexpr result_type min() { return Engine::min; }
static constexpr result_type max() { return Engine::max; }

// constructors and seeding functions
shuffle_order_engine();
explicit shuffle_order_engine(const Engine& e);
explicit shuffle_order_engine(Engine&& e);
explicit shuffle_order_engine(result_type s);
template<SeedSequence class Sseq> explicit shuffle_order_engine(Sseq& q);
void seed();
void seed(result_type s);
template<SeedSequence class Sseq> void seed(Sseq& q);

// generating functions
result_type operator()();
void discard(unsigned long long z);

// property functions
const Engine& base() const;

private:
    Engine e;    // exposition only
    result_type Y; // exposition only
    result_type V[k]; // exposition only
};

4 The following relation shall hold: 0 < k.
5 The textual representation consists of the textual representation of e, followed by the k values of V, followed by the value of Y.
6 In addition to its behavior pursuant to section 26.5.1.5, each constructor that is not a copy constructor initializes V[0],...,V[k-1] and Y, in that order, with values returned by successive invocations of e().

26.5.5 Engines and engine adaptors with predefined parameters [rand.predef]

typedef linear_congruential_engine<uint_fast32_t, 16807, 0, 2147483647>
    minstd_rand0;
    Required behavior: The 10000th consecutive invocation of a default-constructed object of type minstd_rand0 shall produce the value 1043618065.

typedef linear_congruential_engine<uint_fast32_t, 48271, 0, 2147483647>
    minstd_rand;
    Required behavior: The 10000th consecutive invocation of a default-constructed object of type minstd_rand shall produce the value 399268537.

typedef mersenne_twister_engine<uint_fast32_t,
    32,624,397,31,0x9908b0df,11,0xffffffff,7,0x9d2c5680,15,0xefc60000,18,1812433253>
    mt19937;
    Required behavior: The 10000th consecutive invocation of a default-constructed object of type mt19937 shall produce the value 4123659995.

typedef mersenne_twister_engine<uint_fast64_t,
    64,312,156,31,0xb5026f5aa96619e9,29,

§ 26.5.5 Numerics library 857
typedef subtract_with_carry_engine<uint_fast32_t, 24, 10, 24>
ranlux24_base;

Required behavior: The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type \texttt{ranlux24\_base} shall produce the value 7937952.

typedef subtract_with_carry_engine<uint_fast64_t, 48, 5, 12>
ranlux48_base;

Required behavior: The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type \texttt{ranlux48\_base} shall produce the value 61839128582725.

typedef discard_block_engine<ranlux24_base, 223, 23>
ranlux24;

Required behavior: The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type \texttt{ranlux24} shall produce the value 9901578.

typedef discard_block_engine<ranlux48_base, 389, 11>
ranlux48;

Required behavior: The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type \texttt{ranlux48} shall produce the value 249142670248501.

typedef shuffle_order_engine<minstd_rand0,256>
knuth\_b;

Required behavior: The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type \texttt{knuth\_b} shall produce the value 1112339016.

typedef \texttt{implementation-defined}

\texttt{default\_random\_engine};

Required behavior: A concept map \texttt{RandomNumberEngine<default\_random\_engine>}, or equivalent, shall be defined in namespace \texttt{std} so as to provide mappings from free functions to the corresponding member functions.

Remark: The choice of engine type named by this \texttt{typedef} is implementation-defined. [Note: The implementation may select this type on the basis of performance, size, quality, or any combination of such factors, so as to provide at least acceptable engine behavior for relatively casual, inexpert, and/or lightweight use. Because different implementations may select different underlying engine types, code that uses this \texttt{typedef} need not generate identical sequences across implementations. — end note]
26.5.6 Class random_device

A random_device uniform random number generator produces non-deterministic random numbers. A
concept map UniformRandomNumberGenerator random_device shall be defined in namespace std so as to
provide mappings from free functions to the corresponding member functions.

If implementation limitations prevent generating non-deterministic random numbers, the implementation
may employ a random number engine.

```cpp
class random_device
{
public:
  // types
  typedef unsigned int result_type;

  // generator characteristics
  static constexpr result_type min() { return numeric_limits<result_type>::min(); }
  static constexpr result_type max() { return numeric_limits<result_type>::max(); }

  // constructors
  explicit random_device(const string& token = implementation-defined);

  // generating functions
  result_type operator()();

  // property functions
  double entropy() const;

  // no copy functions
  random_device(const random_device& ) = delete;
  void operator=(const random_device& ) = delete;
};
```

**Effects:** Constructs a random_device non-deterministic uniform random number generator object.
The semantics and default value of the token parameter are implementation-defined.\(^{273}\)

**Throws:** A value of an implementation-defined type derived from exception if the random_device
could not be initialized.

```cpp
double entropy() const;
```

**Returns:** If the implementation employs a random number engine, returns 0.0. Otherwise, returns
an entropy estimate\(^{274}\) for the random numbers returned by operator(), in the range min() to
log\(_2\)(max() + 1).

**Throws:** Nothing.

```cpp
result_type operator()();
```

**Returns:** A non-deterministic random value, uniformly distributed between min() and max(), inclusive.
It is implementation-defined how these values are generated.

**Throws:** A value of an implementation-defined type derived from exception if a random number could
not be obtained.

\(^{273}\) The parameter is intended to allow an implementation to differentiate between different sources of randomness.

\(^{274}\) If a device has \(n\) states whose respective probabilities are \(P_0, \ldots, P_{n-1}\), the device entropy \(S\) is defined as
\(S = -\sum_{i=0}^{n-1} P_i \cdot \log P_i.\)
26.5.7 Utilities

26.5.7.1 Class seed_seq

No function described in this section 26.5.7.1 throws an exception.

A concept map SeedSequence<seed_seq> shall be defined in namespace std so as to provide mappings from free functions to the corresponding member functions.

class seed_seq
{
public:
   // types
   typedef uint_least32_t result_type;

   // constructors
   seed_seq();
   template<IntegralLike class T>
   requires IntegralType<T>
   seed_seq(initializer_list<T> il);
   template<class InputIterator Iter>
   requires IntegralLike<Iter::value_type>
   && IntegralType<Iter::value_type>
   seed_seq(Iter InputIterator begin, Iter InputIterator end);

   // generating functions
   template<class RandomAccessIterator Iter>
   requires UnsignedIntegralLike<Iter::value_type>
   && OutputIterator<Iter, const result_type&>
   && IntegralType<Iter::value_type>
   && True<sizeof uint32_t <= sizeof Iter::value_type>
   void generate(Iter RandomAccessIterator begin, Iter RandomAccessIterator end);

   // property functions
   size_t size() const;
   template<class OutputIterator Iter>
   requires OutputIterator<auto, const result_type&>
   void param(Iter OutputIterator dest) const;

   // no copy functions
   seed_seq(const seed_seq&) = delete;
   void operator=(const seed_seq&) = delete;

private:
   vector<result_type> v;  // exposition only
};

Effects: Constructs a seed_seq object as if by default-constructing its member v.

template<IntegralLike class T>
requires IntegralType<T>
seed_seq(initializer_list<T> il);

Requires: T shall be an integer type.

Effects: Same as seed_seq(il.begin(), il.end()).

§ 26.5.7.1 Numerics library 860
requires IntegralLike<Iter::value_type>
   && IntegralType<Iter::value_type>
seed_seq(Iter InputIterator begin, Iter InputIterator end);

Requires: InputIterator shall satisfy the requirements of an input iterator (Table 101) type. Moreover, iterator_traits<InputIterator>::value_type shall denote an integer type.

Effects: Constructs a seed_seq object by the following algorithm:

for( InputIterator s = begin; s != end; ++s)
v.push_back((s)mod2^32);

template<class RandomAccessIterator Iter>
requires UnsignedIntegralLike<Iter::value_type>
   && OutputIterator<Iter, const result_type&>
   && IntegralType<Iter::value_type>
   && True(sizeof uint32_t <= sizeof Iter::value_type>
void generate(Iter RandomAccessIterator begin, Iter RandomAccessIterator end);

Requires: RandomAccessIterator shall meet the requirements of a mutable random access iterator (Table 105) type. Moreover, iterator_traits<RandomAccessIterator>::value_type shall denote an unsigned integer type capable of accommodating 32-bit quantities.

Effects: Does nothing if begin == end. Otherwise, with s = v.size() and n = end − begin, fills the supplied range [begin, end) according to the following algorithm in which each operation is to be carried out modulo 2^{32}, each indexing operator applied to begin is to be taken modulo n, and T(x) is defined as \( x \text{ xor} (x \text{ rshift } 27) \):

a) By way of initialization, set each element of the range to the value 0x8b8b8b8b. Additionally, for use in subsequent steps, let \( p = (n - t)/2 \) and let \( q = p + t \), where

\[
t = (n \geq 623) \mod 11 : (n \geq 68) \mod 7 : (n \geq 39) \mod 5 : (n \geq 7) \mod 3 : (n - 1)/2;
\]

b) With \( m \) as the larger of \( s + 1 \) and \( n \), transform the elements of the range: iteratively for \( k = 0, \ldots, m - 1 \), calculate values

\[
r_1 = 1664525 \cdot T\left(\begin{array}{c}
begin[k] \text{ xor} \begin{array}{c}
begin[k+1] \text{ xor} \begin{array}{c}
begin[k+2]
\end{array}
\end{array}
\end{array}
\end{array}
\right)
\]

\[
r_2 = r_1 + \begin{cases}
   s, & k = 0 \\
   k \mod n \cdot \nu[k - 1], & 0 < k \leq s \\
   k \mod n, & s < k
\end{cases}
\]

and, in order, increment \( \begin{array}{c}
begin[k+1] \end{array} \) by \( r_1 \), increment \( \begin{array}{c}
begin[x+q] \end{array} \) by \( r_2 \), and set \( \begin{array}{c}
begin[k] \end{array} \) to \( r_2 \).

c) Transform the elements of the range three more times, beginning where the previous step ended: iteratively for \( k = m, \ldots, m+n - 1 \), calculate values

\[
r_3 = 1566083941 \cdot T\left(\begin{array}{c}
begin[k] + \begin{array}{c}
begin[k+1] + \begin{array}{c}
begin[k+2]
\end{array}
\end{array}
\end{array}
\end{array}
\right)
\]

\[
r_4 = r_3 - (k \mod n)
\]

and, in order, update \( \begin{array}{c}
begin[k+1] \end{array} \) by xoring it with \( r_4 \), update \( \begin{array}{c}
begin[k+q] \end{array} \) by xoring it with \( r_3 \), and set \( \begin{array}{c}
begin[k] \end{array} \) to \( r_4 \).

size_t size() const;

Returns: The number of 32-bit units that would be returned by a call to \( \text{param}() \).

Complexity: constant time.
template<class OutputIterator<auto, const result_type&>, Iter>
void param(Iter OutputIterator dest) const;

Requires: OutputIterator shall satisfy the requirements of an output iterator (Table 102) type. Moreover, the expression *dest = rt shall be valid for a value rt of type result_type.
Effects: Copies the sequence of prepared 32-bit units to the given destination, as if by executing the following statement:

\[
\text{copy(v.begin(), v.end(), dest);}
\]

26.5.7.2 Function template generate_canonical [rand.util.canonical]

1 Each function instantiated from the template described in this section 26.5.7.2 maps the result of one or more invocations of a supplied uniform random number generator \( g \) to one member of the specified RealType such that, if the values \( g_i \) produced by \( g \) are uniformly distributed, the instantiation’s results \( t_j, 0 \leq t_j < 1 \), are distributed as uniformly as possible as specified below.

2 [Note: Obtaining a value in this way can be a useful step in the process of transforming a value generated by a uniform random number generator into a value that can be delivered by a random number distribution. —end note]

template<
FloatingPointLike class RealType, size_t bits, UniformRandomNumberGenerator<
class URNG>

RealType generate_canonical(URNG& g);

Complexity: Exactly \( k = \max(1, \lceil b / \log_2 R \rceil) \) invocations of \( g \), where \( b^{275} \) is the lesser of numeric_limits<RealType>::digits and bits, and \( R \) is the value of \( g\text{.max()} - g\text{.min()} + 1 \).

Effects: Invokes \( g() \) \( k \) times to obtain values \( g_0, \ldots, g_{k-1} \), respectively. Calculates a quantity

\[
S = \sum_{i=0}^{k-1} (g_i - g\text{.min()}) \cdot R^i
\]

using arithmetic of type RealType.

Returns: \( S/R^k \).

Throws: What and when \( g \) throws.

26.5.8 Random number distribution class templates [rand.dist]

1 Each type instantiated from a class template specified in this section 26.5.8 satisfies the requirements of a random number distribution (26.5.1.6) type.

2 For every class \( D \) specified in this section 26.5.8 or instantiated from a template specified in this section, a concept map RandomNumberDistribution<\( D \)> shall be defined in namespace std as to provide mappings from free functions to the corresponding member functions. Descriptions are provided here in this section 26.5.8 only for distribution operations that are not described in 26.5.1.6 or for operations where there is additional semantic information. In particular, \( D \) declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

3 The algorithms for producing each of the specified distributions are implementation-defined.

4 The value of each probability density function \( p(z) \) and of each discrete probability function \( P(z_i) \) specified in this section is 0 everywhere outside its stated domain.

26.5.8.1 Uniform distributions [rand.dist.uni]

26.5.8.1.1 Class template uniform_int_distribution [rand.dist.uni.int]

1 A uniform_int_distribution random number distribution produces random integers \( i, a \leq i \leq b \), distributed according to the constant discrete probability function

\[
P(i \mid a, b) = 1/(b - a + 1)
\]

\( b \) is introduced to avoid any attempt to produce more bits of randomness than can be held in RealType.

§ 26.5.8.1.1 Numerics library 862
template<IntegralLike class IntType = int>
    requires IntegralType<IntType>
class uniform_int_distribution
{
public:
    // types
    typedef IntType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit uniform_int_distribution(IntType a = 0, IntType b = numeric_limits<IntType>::max());
    explicit uniform_int_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<UniformRandomNumberGenerator class URNG>
        result_type operator()(URNG& g);
    template<UniformRandomNumberGenerator class URNG>
        result_type operator()(URNG& g, const param_type& parm);

    // property functions
    result_type a() const;
    result_type b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};

explicit uniform_int_distribution(IntType a = 0, IntType b = numeric_limits<IntType>::max());

2 Requires: \(a \leq b\).

3 Effects: Constructs a `uniform_int_distribution` object; \(a\) and \(b\) correspond to the respective parameters of the distribution.

result_type a() const;

4 Returns: The value of the \(a\) parameter with which the object was constructed.

result_type b() const;

5 Returns: The value of the \(b\) parameter with which the object was constructed.

26.5.8.1.2 Class template `uniform_real_distribution` [rand.dist.uni.real]

A `uniform_real_distribution` random number distribution produces random numbers \(x, a \leq x < b\), distributed according to the constant probability density function

\[ p(x \mid a, b) = \frac{1}{b-a} \, . \]
typedef RealType result_type;
typedef unspecified param_type;

// constructors and reset functions
explicit uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);
explicit uniform_real_distribution(const param_type& parm);
void reset();

// generating functions
template<typename UniformRandomNumberGenerator>
result_type operator()(UniformRandomNumberGenerator& g);

// property functions
result_type a() const;
result_type b() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;

explicit uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);
2
Requires: a ≤ b and b − a ≤ numeric_limits<RealType>::max().
3
Effects: Constructs a uniform_real_distribution object; a and b correspond to the respective parameters of the distribution.

result_type a() const;
4
Returns: The value of the a parameter with which the object was constructed.

result_type b() const;
5
Returns: The value of the b parameter with which the object was constructed.

26.5.8.2 Bernoulli distributions [rand.dist.bern]
26.5.8.2.1 Class bernoulli_distribution [rand.dist.bern.bernoulli]
1
A bernoulli_distribution random number distribution produces bool values b distributed according to the discrete probability function

\[ P(b \mid p) = \begin{cases} 
    p & \text{if } b = \text{true} \\
    1 - p & \text{if } b = \text{false} 
\end{cases} \]

class bernoulli_distribution
{
public:
    // types
    typedef bool result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit bernoulli_distribution(double p = 0.5);
explicit bernoulli_distribution(const param_type& parm);
void reset();

// generating functions
template<UniformRandomNumberGenerator class URNG>
result_type operator()(URNG& g);
template<UniformRandomNumberGenerator class URNG>
result_type operator()(URNG& g, const param_type& parm);

// property functions
double p() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;
};
26.5.8.2.3 Class template geometric_distribution

A geometric_distribution random number distribution produces integer values $i \geq 0$ distributed according to the discrete probability function

$$P(i | p) = p \cdot (1 - p)^i.$$
explicit geometric_distribution(double p = 0.5);

2 Requires: 0 < p < 1.
3 Effects: Constructs a `geometric_distribution` object; p corresponds to the parameter of the distribution.

double p() const;
4 Returns: The value of the p parameter with which the object was constructed.

26.5.8.2.4 Class template `negative_binomial_distribution` [rand.dist.bern.negbin]
1 A `negative_binomial_distribution` random number distribution produces random integers \( i \geq 0 \) distributed according to the discrete probability function

\[
P(i | k, p) = \binom{k + i - 1}{i} p^k (1 - p)^i.
\]

```cpp
template<IntegralLike class IntType = int>
    requires IntegralType<IntType>
class negative_binomial_distribution
{
public:
    // types
    typedef IntType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit negative_binomial_distribution(IntType k = 1, double p = 0.5);
    explicit negative_binomial_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<UniformRandomNumberGenerator class URNG>
    result_type operator()(URNG& g);
    template<UniformRandomNumberGenerator class URNG>
    result_type operator()(URNG& g, const param_type& parm);

    // property functions
    IntType k() const;
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

explicit negative_binomial_distribution(IntType k = 1, double p = 0.5);

2 Requires: 0 < p ≤ 1 and 0 < k.
3 Effects: Constructs a `negative_binomial_distribution` object; k and p correspond to the respective parameters of the distribution.

IntType k() const;
4 Returns: The value of the k parameter with which the object was constructed.

§ 26.5.8.2.4 Numerics library 867
double p() const;

Returns: The value of the p parameter with which the object was constructed.

26.5.8.3 Poisson distributions

26.5.8.3.1 Class template poisson_distribution

A poisson_distribution random number distribution produces integer values \( i \geq 0 \) distributed according to the discrete probability function
\[
P(i | \mu) = \frac{e^{-\mu} \mu^i}{i!}.
\]
The distribution parameter \( \mu \) is also known as this distribution’s mean.

\[
template<\text{IntegralLike} class \text{IntType} = \text{int}>
\text{requires IntegralType<IntType>}
\text{class poisson_distribution}
{
\text{public:}
// types
typedef IntType result_type;
typedef unspecified param_type;

// constructors and reset functions
\text{explicit poisson_distribution(double mean = 1.0);}
\text{explicit poisson_distribution(const param_type& parm);}
\text{void reset();}

// generating functions
\text{template<UniformRandomNumberGenerator class URNG>}
\text{result_type operator()}(URNG& g);
\text{template<UniformRandomNumberGenerator class URNG>}
\text{result_type operator()}(URNG& g, const param_type& parm);

// property functions
\text{double mean() const;}
\text{param_type param() const;}
\text{void param(const param_type& parm);}
\text{result_type min() const;}
\text{result_type max() const;}
};

\text{explicit poisson_distribution(double mean = 1.0);}

Requires: 0 < mean.
Effects: Constructs a poisson_distribution object; mean corresponds to the parameter of the distribution.

double mean() const;

Returns: The value of the mean parameter with which the object was constructed.

26.5.8.3.2 Class template exponential_distribution

An exponential_distribution random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function
\[
p(x | \lambda) = \lambda e^{-\lambda x}.
\]
template<
    FloatingPointLike::class RealType = double>
requires FloatingPointType<RealType>
class exponential_distribution
{
public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit exponential_distribution(RealType lambda = 1.0);
    explicit exponential_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<
        UniformRandomNumberGenerator::class URNG>
    result_type operator()(URNG& g);
    template<
        UniformRandomNumberGenerator::class URNG>
    result_type operator()(URNG& g, const param_type& parm);

    // property functions
    RealType lambda() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};

explicit exponential_distribution(RealType lambda = 1.0);

2
Requires: 0 < lambda.
3
Effects: Constructs a exponential_distribution object; lambda corresponds to the parameter of the
distribution.

RealType lambda() const;

4
    Returns: The value of the lambda parameter with which the object was constructed.

26.5.8.3.3 Class template gamma_distribution
A gamma_distribution random number distribution produces random numbers \( x > 0 \) distributed according
to the probability density function
\[
p(x \mid \alpha, \beta) = \frac{e^{-x/\beta}}{\beta^{\alpha} \cdot \Gamma(\alpha)} \cdot x^{\alpha-1}.
\]

template<
    FloatingPointLike::class RealType = double>
requires FloatingPointType<RealType>
class gamma_distribution
{
public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
explicit gamma_distribution(RealType alpha = 1.0, RealType beta = 1.0);
explicit gamma_distribution(const param_type& parm);
void reset();

// generating functions
template<UniformRandomNumberGenerator class URNG>
result_type operator()(URNG& g);
template<UniformRandomNumberGenerator class URNG>
result_type operator()(URNG& g, const param_type& parm);

// property functions
RealType alpha() const;
RealType beta() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;


explicit gamma_distribution(RealType alpha = 1.0, RealType beta = 1.0);

Requires: 0 < alpha and 0 < beta.
Effects: Constructs a gamma_distribution object; alpha and beta correspond to the parameters of the distribution.

RealType alpha() const;
Returns: The value of the alpha parameter with which the object was constructed.

RealType beta() const;
Returns: The value of the beta parameter with which the object was constructed.

26.5.8.3.4 Class template weibull_distribution

A weibull_distribution random number distribution produces random numbers $x \geq 0$ distributed according to the probability density function

$$p(x | a, b) = \frac{a}{b} \cdot \left( \frac{x}{b} \right)^{a-1} \cdot \exp \left( -\left( \frac{x}{b} \right)^a \right).$$

template<FloatingPointLike class RealType = double>
requires FloatingPointType<RealType>
class weibull_distribution
{
public:

// types
typedef RealType result_type;
typedef unspecified param_type;

// constructor and reset functions
explicit weibull_distribution(RealType a = 1.0, RealType b = 1.0)
explicit weibull_distribution(const param_type& parm);
void reset();

// generating functions
template<UniformRandomNumberGenerator class URNG>
  result_type operator()(URNG& g);

template<UniformRandomNumberGenerator class URNG>
  result_type operator()(URNG& g, const param_type& parm);

// property functions
RealType a() const;
RealType b() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;
};

explicit weibull_distribution(RealType a = 1.0, RealType b = 1.0);

Requires: 0 < a and 0 < b.
Effects: Constructs a weibull_distribution object; a and b correspond to the respective parameters of the distribution.

RealType a() const;
Returns: The value of the a parameter with which the object was constructed.

RealType b() const;
Returns: The value of the b parameter with which the object was constructed.

26.5.8.3.5 Class template extreme_value_distribution [rand.dist.pois.extreme]

An extreme_value_distribution random number distribution produces random numbers x distributed according to the probability density function\(^{276}\)

\[
p(x \mid a, b) = \frac{1}{b} \cdot \exp\left(\frac{a - x}{b} - \exp\left(\frac{a - x}{b}\right)\right).
\]

\(^{276}\) The distribution corresponding to this probability density function is also known (with a possible change of variable) as the Gumbel Type I, the log-Weibull, or the Fisher-Tippett Type I distribution.
template<UniformRandomNumberGenerator class URNG>
  result_type operator()(URNG& g, const param_type& parm);

// property functions
  RealType a() const;
  RealType b() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
};

explicit extreme_value_distribution(RealType a = 0.0, RealType b = 1.0);

Requires: 0 < b.
Effects: Constructs an extreme_value_distribution object; a and b correspond to the respective parameters of the distribution.

RealType a() const;
Returns: The value of the a parameter with which the object was constructed.

RealType b() const;
Returns: The value of the b parameter with which the object was constructed.

26.5.8.4 Normal distributions

26.5.8.4.1 Class template normal_distribution

A normal_distribution random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x | \mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} \exp \left( -\frac{(x - \mu)^2}{2\sigma^2} \right).
\]

The distribution parameters \( \mu \) and \( \sigma \) are also known as this distribution's mean and standard deviation.

template<FloatingPointLike class RealType = double>
requires FloatingPointType<RealType>
  class normal_distribution
  {
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit normal_distribution(RealType mean = 0.0, RealType stddev = 1.0);
    explicit normal_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<UniformRandomNumberGenerator class URNG>
    result_type operator()(URNG& g);
    template<UniformRandomNumberGenerator class URNG>
    result_type operator()(URNG& g, const param_type& parm);

  § 26.5.8.4.1 Numerics library
// property functions
RealType mean() const;
RealType stddev() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;
};

explicit normal_distribution(RealType mean = 0.0, RealType stddev = 1.0);

Requires: 0 < stddev.
Effects: Constructs a normal_distribution object; mean and stddev correspond to the respective parameters of the distribution.

RealType mean() const;
Returns: The value of the mean parameter with which the object was constructed.

RealType stddev() const;
Returns: The value of the stddev parameter with which the object was constructed.

26.5.8.4.2 Class template lognormal_distribution

A lognormal_distribution random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$ p(x | m, s) = \frac{1}{sx\sqrt{2\pi}} \cdot \exp \left( -\frac{(\ln x - m)^2}{2s^2} \right). $$

template<FloatingPointLike class RealType = double>
requires FloatingPointType<RealType>
class lognormal_distribution
{
public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);
    explicit lognormal_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<UniformRandomNumberGenerator class URNG>
    result_type operator()(URNG& g);
    template<UniformRandomNumberGenerator class URNG>
    result_type operator()(URNG& g, const param_type& parm);

    // property functions
    RealType m() const;
    RealType s() const;
    param_type param() const;

§ 26.5.8.4.2 Numerics library
void param(const param_type& parm);
result_type min() const;
result_type max() const;
};

explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);

Requires: 0 < s.
Effects: Constructs a lognormal_distribution object; m and s correspond to the respective parameters of the distribution.

RealType m() const;
Returns: The value of the m parameter with which the object was constructed.

RealType s() const;
Returns: The value of the s parameter with which the object was constructed.

26.5.8.4.3 Class template chi_squared_distribution [rand.dist.norm.chisq]

A chi_squared_distribution random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[
p(x | n) = \frac{x^{n/2-1} \cdot e^{-x/2}}{\Gamma(n/2) \cdot 2^{n/2}}.
\]

```cpp
template<FloatingPointLike RealType = double>
requires FloatingPoint<RealType>
class chi_squared_distribution
{
    public:
        // types
        typedef RealType result_type;
        typedef unspecified param_type;

        // constructor and reset functions
        explicit chi_squared_distribution(RealType n = 1);
        explicit chi_squared_distribution(const param_type& parm);
        void reset();

        // generating functions
        template<UniformRandomNumberGenerator class URNG>
        result_type operator()(URNG& g);
        template<UniformRandomNumberGenerator class URNG>
        result_type operator()(URNG& g, const param_type& parm);

        // property functions
        RealType n() const;
        param_type param() const;
        void param(const param_type& parm);
        result_type min() const;
        result_type max() const;
};

explicit chi_squared_distribution(RealType n = 1);
```
2 \textit{Requires:} \(0 < n\).
3 \textit{Effects:} Constructs a \texttt{chi\_squared\_distribution} object; \(n\) corresponds to the parameter of the distribution.

\texttt{RealType n() const;}
4 \textit{Returns:} The value of the \(n\) parameter with which the object was constructed.

26.5.8.4.4 \textbf{Class template \texttt{cauchy\_distribution}} \texttt{[rand.dist.norm.cauchy]}

1 A \texttt{cauchy\_distribution} random number distribution produces random numbers \(x\) distributed according to the probability density function

\[
p(x | a, b) = \frac{\pi b}{\pi b \left(1 + \left(\frac{x - a}{b}\right)^2\right)}.
\]

template<\texttt{FloatingPointLike} class RealType = double>
\texttt{requires FloatingPointType<RealType>}
\texttt{class cauchy\_distribution}
\{
public:

// types
\texttt{typedef RealType result\_type;}
\texttt{typedef unspecified param\_type;}

// constructor and reset functions
\texttt{explicit cauchy\_distribution(RealType a = 0.0, RealType b = 1.0);}  
\texttt{explicit cauchy\_distribution(const param\_type& parm);}  
\texttt{void reset();}

// generating functions
\texttt{template<\texttt{UniformRandomNumberGenerator} class URNG>}
\texttt{result\_type operator()\(\texttt{(URNG}\& g);\)}
\texttt{template<\texttt{UniformRandomNumberGenerator} class URNG>}
\texttt{result\_type operator()\(\texttt{(URNG}\& g, \texttt{const param\_type}\&\texttt{ parm});\)}

// property functions
\texttt{RealType a() const;}
\texttt{RealType b() const;}
\texttt{param\_type param() const;}
\texttt{void param(const param\_type& parm);}
\texttt{result\_type min() const;}
\texttt{result\_type max() const;}
\};

\texttt{explicit cauchy\_distribution(RealType a = 0.0, RealType b = 1.0);}  
2 \textit{Requires:} \(0 < b\).
3 \textit{Effects:} Constructs a \texttt{cauchy\_distribution} object; \(a\) and \(b\) correspond to the respective parameters of the distribution.

\texttt{RealType a() const;}
4 \textit{Returns:} The value of the \(a\) parameter with which the object was constructed.
RealType b() const;

Returns: The value of the $b$ parameter with which the object was constructed.

26.5.8.4.5 Class template fisher_f_distribution

A fisher_f_distribution random number distribution produces random numbers $x \geq 0$ distributed according to the probability density function

$$p(x \mid m, n) = \frac{\Gamma((m + n)/2)}{\Gamma(m/2) \Gamma(n/2)} \left(\frac{m}{n}\right)^{m/2} x^{(m/2)-1} \left(1 + \frac{mx}{n}\right)^{-(m+n)/2}.$$ 

template<
    FloatingPointLike <class RealType = double>
    requires FloatingPointType<RealType>
>
    class fisher_f_distribution
    {
    public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit fisher_f_distribution(RealType m = 1, RealType n = 1);
    explicit fisher_f_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<
        UniformRandomNumberGenerator <class URNG>
    >
        result_type operator()(URNG& g);
    template<
        UniformRandomNumberGenerator <class URNG>
    >
        result_type operator()(URNG& g, const param_type& parm);

    // property functions
    RealType m() const;
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
    
    explicit fisher_f_distribution(RealType m = 1, RealType n = 1);

    Requires: $0 < m$ and $0 < n$.

    Effects: Constructs a fisher_f_distribution object; $m$ and $n$ correspond to the respective parameters of the distribution.

    RealType m() const;

    Returns: The value of the $m$ parameter with which the object was constructed.

    RealType n() const;

    Returns: The value of the $n$ parameter with which the object was constructed.
26.5.8.4.6 Class template \texttt{student\_t\_distribution} [rand.dist.norm.t]

A \texttt{student\_t\_distribution} random number distribution produces random numbers $x$ distributed according to the probability density function

$$p(x \mid n) = \frac{1}{\sqrt{n\pi}} \frac{\Gamma((n+1)/2)}{\Gamma(n/2)} \left(1 + \frac{x^2}{n}\right)^{-(n+1)/2}.$$

\begin{verbatim}
#include <random>

template<
    FloatingPointLike class RealType = double>
require FloatingPointType<RealType>

class student_t_distribution
{
public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit student_t_distribution(RealType n = 1);
    explicit student_t_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<
        UniformRandomNumberGenerator class URNG>
    result_type operator()(URNG& g);
    template<
        UniformRandomNumberGenerator class URNG>
    result_type operator()(URNG& g, const param_type& parm);

    // property functions
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
\end{verbatim}

\begin{verbatim}
explicit student_t_distribution(RealType n = 1);
\end{verbatim}

\textbf{Requires:} $0 < n$.

\textbf{Effects:} Constructs a \texttt{student\_t\_distribution} object; $n$ corresponds to the parameter of the distribution.

\textbf{Returns:} The value of the $n$ parameter with which the object was constructed.

26.5.8.5 Sampling distributions [rand.dist.samp]

26.5.8.5.1 Class template \texttt{discrete\_distribution} [rand.dist.samp.discrete]

A \texttt{discrete\_distribution} random number distribution produces random integers $i$, $0 \leq i < n$, distributed according to the discrete probability function

$$P(i \mid p_0, \ldots, p_{n-1}) = p_i.$$

\begin{verbatim}
#include <random>

template<
    FloatingPointLike class RealType = double>
require FloatingPointType<RealType>

class discrete_distribution
{
public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit discrete_distribution(RealType p);
    explicit discrete_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<
        UniformRandomNumberGenerator class URNG>
    result_type operator()(URNG& g);
    template<
        UniformRandomNumberGenerator class URNG>
    result_type operator()(URNG& g, const param_type& parm);

    // property functions
    RealType param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
\end{verbatim}

\begin{verbatim}
explicit discrete_distribution(RealType p);
\end{verbatim}

\textbf{Requires:} $0 < n$.

\textbf{Effects:} Constructs a \texttt{discrete\_distribution} object; $n$ corresponds to the parameter of the distribution.

\textbf{Returns:} The value of the $n$ parameter with which the object was constructed.

Unless specified otherwise, the distribution parameters are calculated as: $p_k = w_k / S$ for $k = 0, \ldots, n-1$, in which the values $w_k$, commonly known as the \textit{weights}, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold: $0 < S = w_0 + \cdots + w_{n-1}$.
template<IntegralLike class IntType = int>
  requires IntegralType<IntType>
class discrete_distribution
{
  public:
    // types
    typedef IntType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    discrete_distribution();
    template<class InputIterator Iter>
      requires Convertible<Iter::value_type, double>
      discrete_distribution(InputIterator firstW, InputIterator lastW);
    discrete_distribution(initializer_list<double> wl);
    template<Callable<auto, double> Func class UnaryOperation>
      requiresConvertible<Func::result_type, double>
      discrete_distribution(size_t nw, double xmin, double xmax, Func UnaryOperation fw);
    explicit discrete_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<UniformRandomNumberGenerator class URNG>
      result_type operator()(URNG& g);
    template<UniformRandomNumberGenerator class URNG>
      result_type operator()(URNG& g, const param_type& parm);

    // property functions
    vector<double> probabilities() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};

discrete_distribution();
  Effects: Constructs a discrete_distribution object with \( n = 1 \) and \( p_0 = 1 \). [Note: Such an object will always deliver the value 0. —end note]

template<class InputIterator Iter>
  requires Convertible<Iter::value_type, double>
discrete_distribution(InputIterator firstW, InputIterator lastW);
  Requires: InputIterator shall satisfy the requirements of an input iterator (Table 101) type. Moreover, iterator_traits<InputIterator>::value_type shall denote a type that is convertible to double. If \( \text{firstW} == \text{lastW} \), let \( n = 1 \) and \( w_0 = 1 \). Otherwise, \([\text{firstW}, \text{lastW})\) shall form a sequence \( w \) of length \( n > 0 \).
  Effects: Constructs a discrete_distribution object with probabilities given by the formula above.

discrete_distribution(initializer_list<double> wl);
  Effects: Same as discrete_distribution(wl.begin(), wl.end()).

template<Callable<auto, double> Func class UnaryOperation>

§ 26.5.8.5.1 Numerics library 878
discrete_distribution(size_t nw, double xmin, double xmax, FuncUnaryOperation fw);

Requires: Each instance of type UnaryOperation shall be a function object (20.7) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation’s sole parameter. If nw = 0, let n = 1, otherwise let n = nw. The relation 0 < δ = (xmax − xmin)/n shall hold.

Effects: Constructs a discrete_distribution object with probabilities given by the formula above, using the following values: If nw = 0, let w₀ = 1. Otherwise, let wₖ = fw(xmin + k · δ + δ/2) for k = 0,...,n−1.

Complexity: The number of invocations of fw shall not exceed n.

vector<double> probabilities() const;

Returns: A vector<double> whose size member returns n and whose operator[] member returns pᵢ when invoked with argument k for k = 0,...,n−1.

26.5.8.5.2 Class template piecewise_constant_distribution

A piecewise_constant_distribution random number distribution produces random numbers x, b₀ ≤ x < bₙ, uniformly distributed over each subinterval [bᵢ, bᵢ₊₁) according to the probability density function

\[ p(x | b₀, ..., bₙ, ρ₀, ..., ρₙ−₁) = ρᵢ, \quad \text{for } bᵢ ≤ x < bᵢ₊₁. \]

The n + 1 distribution parameters bᵢ, also known as this distribution’s internal boundaries, shall satisfy the relation bᵢ < bᵢ₊₁ for i = 0,...,n−1. Unless specified otherwise, the remaining n distribution parameters are calculated as:

\[ ρ_k = \frac{w_k}{S · (b_k+1 − b_k)} \quad \text{for } k = 0,...,n−1, \]

in which the values wₖ, commonly known as the weights, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold: 0 < S = w₀ + ⋯ + wₙ−₁.

template<FloatingPointLike class RealType = double>
requires FloatingPointType<RealType>
class piecewise_constant_distribution
{
    public:
        // types
        typedef RealType result_type;
        typedef unspecified param_type;

        // constructor and reset functions
        piecewise_constant_distribution();
        template<InputIterator IterB, InputIterator IterW>
            requires Convertible<IterB::value_type, result_type> & Convertible<IterW::value_type, double>
            piecewise_constant_distribution(IterB firstB, IterB lastB, IterW firstW);
        template<Callable<auto, RealType> Func>
            requires Convertible<Func::result_type, double>
            piecewise_constant_distribution(initializer_list<RealType> bl, Func unaryOperation fw);
        template<Callable<auto, double> Func>
            requires Convertible<Func::result_type, double>
            piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax, Func unaryOperation fw);
        explicit piecewise_constant_distribution(const param_type& parm);
        void reset();

§ 26.5.8.5.2 Numerics library
template<UniformRandomNumberGenerator class URNG>
result_type operator()(URNG& g);

// property functions
vector<RealType> intervals() const;
vector<double> densities() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;

piecewise_constant_distribution();

Effects: Constructs a piecewise_constant_distribution object with \( n = 1, \rho_0 = 1, b_0 = 0, \) and \( b_1 = 1. \)

template<InputIterator_IterB, InputIterator_IterW class InputIteratorB, class InputIteratorW>
requires Convertible<IterB::value_type, result_type> && Convertible<IterW::value_type, double>
piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB,
InputIteratorW firstW);

Requires: InputIteratorB and InputIteratorW shall each satisfy the requirements of an input iterator (Table 101) type. Moreover, iterator_traits<InputIteratorB>::value_type and iterator_traits<InputIteratorW>::value_type shall each denote a type that is convertible to double.

If firstB == lastB or ++firstB == lastB, let \( n = 1, w_0 = 1, b_0 = 0, \) and \( b_1 = 1. \) Otherwise, [firstB, lastB) shall form a sequence \( b \) of length \( n + 1, \) the length of the sequence \( w \) starting from firstW shall be at least \( n, \) and any \( w_k \) for \( k \geq n \) shall be ignored by the distribution.

Effects: Constructs a piecewise_constant_distribution object with parameters as specified above.

template<Callable<auto, RealType> Func class UnaryOperation>
requires Convertible<Func::result_type, double>
piecewise_constant_distribution(initializer_list<RealType> bl, Func UnaryOperation fw);

Requires: Each instance of type UnaryOperation shall be a function object (20.7) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation's sole parameter.

Effects: Constructs a piecewise_constant_distribution object with parameters taken or calculated from the following values: If \( bl.\text{size}() < 2, \) let \( n = 1, w_0 = 1, b_0 = 0, \) and \( b_1 = 1. \) Otherwise, let [bl.begin(), bl.end()] form a sequence \( b_0, \ldots, b_n, \) and let \( w_k = fw((b_{k+1}+b_k)/2) \) for \( k = 0, \ldots, n-1. \)

Complexity: The number of invocations of \( fw \) shall not exceed \( n. \)

template<Callable<auto, double> Func class UnaryOperation>
requires Convertible<Func::result_type, double>
piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax, Func UnaryOperation fw);

Requires: Each instance of type UnaryOperation shall be a function object (20.7) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation's sole parameter. If \( nw = 0, \) let \( n = 1, \) otherwise let \( n = nw. \) The relation \( 0 < \delta = (xmax - xmin)/n \) shall hold.

Effects: Constructs a piecewise_constant_distribution object with parameters taken or calculated from the following values: Let \( b_k = xmin + k\cdot\delta \) for \( k = 0, \ldots, n, \) and \( w_k = fw(b_k+\delta/2) \) for \( k = 0, \ldots, n-1. \)

Complexity: The number of invocations of \( fw \) shall not exceed \( n. \)
vector<result_type> intervals() const;

Returns: A vector<result_type> whose size member returns \( n + 1 \) and whose operator[] member returns \( b_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n \).

vector<double> densities() const;

Returns: A vector<result_type> whose size member returns \( n \) and whose operator[] member returns \( \rho_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n-1 \).

26.5.8.5.3 Class template piecewise_linear_distribution  

A piecewise_linear_distribution random number distribution produces random numbers \( x, b_0 \leq x < b_n \), distributed over each subinterval \( [b_i, b_{i+1}) \) according to the probability density function

\[
p(x | b_0, \ldots, b_n, \rho_0, \ldots, \rho_n) = \rho_i \cdot \frac{x - b_i b_{i+1} - x}{b_{i+1} - b_i} + \rho_{i+1} \cdot \frac{b_{i+1} - x - b_i}{b_{i+1} - b_i}, \text{ for } b_i \leq x < b_{i+1}.
\]

The \( n + 1 \) distribution parameters \( b_i \), also known as this distribution’s interval boundaries, shall satisfy the relation \( b_i < b_{i+1} \) for \( i = 0, \ldots, n-1 \). Unless specified otherwise, the remaining \( n + 1 \) distribution parameters are calculated as \( \rho_k = w_k / S \) for \( k = 0, \ldots, n \), in which the values \( w_k \), commonly known as the weights at boundaries, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold:

\[
0 < S = \frac{1}{2} \cdot \sum_{k=0}^{n-1} (\rho w_k + \rho w_{k+1}) \cdot (b_{k+1} - b_k).
\]

```cpp
template<FloatingPointLike class RealType = double>
requires FloatingPointType<RealType>
class piecewise_linear_distribution 
{
public:
  // types
  typedef RealType result_type;
  typedef unspecified param_type;

  // constructor and reset functions
  piecewise_linear_distribution();

template<InputIterator IterB, InputIterator IterW class InputIteratorB, class InputIteratorW>
  requires Convertible<IterB::value_type, result_type> && Convertible<IterW::value_type, double>
  piecewise_linear_distribution(IterB InputIteratorB firstB, IterB InputIteratorB lastB,
                            IterW InputIteratorW firstW);

template<Callable<auto, RealType> Func class UnaryOperation>
  requires Convertible<Func::result_type, double>
  piecewise_linear_distribution(initializer_list<RealType> bl, Func UnaryOperation fw);

template<Callable<auto, double> Func UnaryOperation>
  requires Convertible<Func::result_type, double>
  piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, Func UnaryOperation fw);

  explicit piecewise_linear_distribution(const param_type& parm);
  void reset();

  // generating functions
  template<UniformRandomNumberGenerator class URNG>
  result_type operator()(URNG& g);

template<UniformRandomNumberGenerator class URNG>
  result_type operator()(URNG& g, const param_type& parm);
```
// property functions
vector<RealType> intervals() const;
vector<double> densities() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;

};
piecewise_linear_distribution();

Effects: Constructs a piecewise_linear_distribution object with \(n = 1, \rho_0 = \rho_1 = 1, b_0 = 0\), and \(b_1 = 1\).

template<input_iterator InputIteratorB, input_iterator InputIteratorW>
requires convertible<InputIteratorB::value_type, result_type> && convertible<InputIteratorW::value_type, double>
piecewise_linear_distribution(InputIteratorB firstB, InputIteratorB lastB,
InputIteratorW firstW);

Requires: \(\text{InputIteratorB}\) and \(\text{InputIteratorW}\) shall each satisfy the requirements of an input iterator (Table 101) type. Moreover, \(\text{iterator_traits<InputIteratorB>::value_type}\) and \(\text{iterator_traits<InputIteratorW>::value_type}\) shall each denote a type that is convertible to \(\text{double}\). If \(\text{firstB} == \text{lastB}\) or ++\(\text{firstB}\) == \(\text{lastB}\), let \(n = 1, \rho_0 = \rho_1 = 1, b_0 = 0\), and \(b_1 = 1\). Otherwise, \([\text{firstB}, \text{lastB}]\) shall form a sequence \(b\) of length \(n + 1\), the length of the sequence \(w\) starting from \(\text{firstW}\) shall be at least \(n + 1\), and any \(w_k\) for \(k \geq n + 1\) shall be ignored by the distribution.

Effects: Constructs a piecewise_linear_distribution object with parameters as specified above.

template<Callable<auto, RealType> Func>
requires convertible<Func::result_type, double>
piecewise_linear_distribution(initializer_list<RealType> bl, Func UnaryOperation fw);

Requires: Each instance of type \(\text{UnaryOperation}\) shall be a function object (20.7) whose return type shall be convertible to \(\text{double}\). Moreover, \(\text{double}\) shall be convertible to the type of \(\text{UnaryOperation}\)'s sole parameter.

Effects: Constructs a piecewise_linear_distribution object with parameters taken or calculated from the following values: If \(\text{bl.size()} < 2\), let \(n = 1, \rho_0 = \rho_1 = 1, b_0 = 0\), and \(b_1 = 1\). Otherwise, let \([\text{bl.begin()}, \text{bl.end()}]\) form a sequence \(b_0, \ldots, b_n\), and let \(w_k = \text{fw}(b_k)\) for \(k = 0, \ldots, n\).

Complexity: The number of invocations of \(\text{fw}\) shall not exceed \(n + 1\).

template<Callable<auto, double> Func>
requires convertible<Func::result_type, double>
piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, Func UnaryOperation fw);

Requires: Each instance of type \(\text{UnaryOperation}\) shall be a function object (20.7) whose return type shall be convertible to \(\text{double}\). Moreover, \(\text{double}\) shall be convertible to the type of \(\text{UnaryOperation}\)'s sole parameter. If \(\text{nw} = 0\), let \(n = 1\), otherwise let \(n = \text{nw}\). The relation \(0 < \delta = (\text{xmax} - \text{xmin})/n\) shall hold.

Effects: Constructs a piecewise_linear_distribution object with parameters taken or calculated from the following values: Let \(b_k = \text{xmin} + k \cdot \delta\) for \(k = 0, \ldots, n\), and \(w_k = \text{fw}(b_k + \delta)\) for \(k = 0, \ldots, n\).

Complexity: The number of invocations of \(\text{fw}\) shall not exceed \(n + 1\).

vector<ResultType> intervals() const;
Returns: A `vector<result_type>` whose `size` member returns \( n + 1 \) and whose `operator[]` member returns \( b_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n \).

`vector<double> densities() const;`

Returns: A `vector<result_type>` whose `size` member returns \( n \) and whose `operator[]` member returns \( \rho_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n \).

### 26.6 Numeric arrays

#### 26.6.1 Header `<valarray>` synopsis

```cpp
namespace std {
    #include <initializer_list>

template<class T> class valarray;  // An array of type T
class slice;  // a BLAS-like slice out of an array
template<class T> class slice_array;
class gslice;  // a generalized slice out of an array
template<class T> class gslice_array;
template<class T> class mask_array;  // a masked array
template<class T> class indirect_array;  // an indirection array

template<class T> void swap(valarray<T>&, valarray<T>&);

template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator* (const valarray<T>&, const T&);
template<class T> valarray<T> operator* (const T&, const valarray<T>&);

template<class T> valarray<T> operator/ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator/ (const valarray<T>&, const T&);
template<class T> valarray<T> operator/ (const T&, const valarray<T>&);

template<class T> valarray<T> operator% (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&, const T&);
template<class T> valarray<T> operator% (const T&, const valarray<T>&);

template<class T> valarray<T> operator+ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&, const T&);
template<class T> valarray<T> operator+ (const T&, const valarray<T>&);

template<class T> valarray<T> operator- (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator- (const valarray<T>&, const T&);
template<class T> valarray<T> operator- (const T&, const valarray<T>&);

template<class T> valarray<T> operator^ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator^ (const valarray<T>&, const T&);
template<class T> valarray<T> operator^ (const T&, const valarray<T>&);

template<class T> valarray<T> operator& (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&, const T&);
template<class T> valarray<T> operator& (const T&, const valarray<T>&);

template<class T> valarray<T> operator| (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&, const T&);
template<class T> valarray<T> operator| (const T&, const valarray<T>&);
```

§ 26.6.1 Numerics library 883