Document Number: N3037=10-0027
Date: 2010-02-11
Reply to: Walter E. Brown <wb@fnal.gov>
FPE Dept., Computing Division
Fermi National Accelerator Laboratory
Batavia, IL 60510-0500
USA

Conceptless Random Number Generation in C++0X

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 (even for some that we believe ought be closed as NAD, in order to provide a basis for discussion). The usual color conventions are used throughout the text to denote the proposed changes, all of which are relative to Working Draft N3000.

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);

template<class RandomAccessIterator>
  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);
```

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

5 *Requires:* The ranges `[first,last)` and `[result,result+(last-first))` shall not overlap.

6 *Returns:* `result + (last - first)`.

7 *Complexity:* Exactly `last - first` assignments.

25.3.11 Rotate [alg.rotate]

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

1 *Effects:* For each non-negative integer `i < (last - first)`, places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`.

2 *Returns:* `first + (last - middle)`.

3 *Remarks:* This is a left rotate.

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

5 *Complexity:* At most `last - first` swaps.

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

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

7 *Returns:* `result + (last - first)`.

8 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

9 *Complexity:* Exactly `last - first` assignments.

25.3.12 Random shuffle [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);
```

```
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 g~~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 [alg.partitions]

```
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`.

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

4 *Effects:* Places all the elements in the range `[first,last)` that satisfy `pred` before all the elements that do not satisfy it.

5 *Returns:* An iterator `i` such that for any iterator `j` in the range `[first,i) pred(*j) != false`, and for any iterator `k` in the range `[i,last)`, `pred(*k) == false`.

6 *Requires:* The type of `*first` shall satisfy the Swappable requirements (37).

7 *Complexity:* If ForwardIterator meets the requirements for a BidirectionalIterator, at most `(last - first) / 2` swaps are done; otherwise at most `last - first` swaps are done. Exactly `last - first` applications of the predicate are done.

26.4.9 Additional Overloads [cmplx.over]

1 The following function templates shall have additional overloads:

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

2 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>`.

3 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> [ccmplx]

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

26.4.11 Header <complex.h> [cmplxh]

1 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 [rand]

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

2 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 ~~concepts~~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*]

3 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`.

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

5 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 `<random>` synopsis, formerly located here, has been moved and now follows the Requirements section below.]

26.5.1 Concepts and related rRequirements for random number generation [rand.req]

26.5.1.1 General requirements [rand.req.genl]

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

a) that has a template type parameter named `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 `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 `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 `RealType` is undefined unless the corresponding template argument is cv-unqualified and is one of `float`, `double`, or `long double`.

e) that has a template type parameter named `IntType` is undefined unless the corresponding template argument is cv-unqualified and is one of `short`, `int`, `long`, `long long`, `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`.

f) that has a template type parameter named `UIntType` is undefined unless the corresponding template argument is cv-unqualified and is one of `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`.

2 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."

3 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 `explicit`.

26.5.1.2 Concept SeedSequenceSeed sequence requirements [rand.req.seedseq]

1 A *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. [*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*]

```
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>
```

```
          && 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 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

Expression	Return type	Pre/post-condition	Complexity
`S::result_type`	`T`	`T` is an unsigned integer type (3.9.1) of at least 32 bits.	compile-time
`S()`	—	Creates a seed sequence with the same initial state as all other default-constructed seed sequences of type `S`.	constant
`S(ib,ie)`	—	Creates a seed sequence having internal state that depends on some or all of the bits of the supplied sequence [`ib`, `ie`).	$\mathcal{O}$(`ie` − `ib`)
`S(il)`	—	Same as `S(il.begin(), il.end())`.	same as `S(il.begin(), il.end())`
`q.generate(rb,re)`	`void`	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.	$\mathcal{O}$(`re` − `rb`)
`r.size()`	`size_t`	The number of 32-bit units that would be copied by a call to `r.param`.	constant
`r.param(ob)`	`void`	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.	$\mathcal{O}$(`r.size()`)

26.5.1.3 Concept `UniformRandomNumberGenerator` Uniform random number generator requirements [rand.req.urng]

1 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 110 — Uniform random number generator requirements

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

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

26.5.1.4 Concept `RandomNumberEngine`Random number engine requirements [rand.req.eng]

1 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 $\mathcal{O}$(size of state).

```
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 $\mathtt{e}_i$ for some integer $i \geq 0$. Upon construction, `e` has an initial state $\mathtt{e}_0$. 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 `e`'s state $\mathtt{e}_i$ is advanced to its *successor state* $\mathtt{e}_{i+1}$; 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 $S_1 = S_2$, where S_1 and S_2 are the infinite sequences of values that would be generated, respectively, by repeated future calls to `e1()` and `e2()`. Otherwise returns `false`.

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

6 *Effects:* Sets `e`'s state to $\mathtt{e}_{i+1} = \mathsf{TA}(\mathtt{e}_i)$.

7 *Returns:* $\mathsf{GA}(\mathtt{e}_i)$.

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()`.

```
void seed(E& e, result_type s);
```

Postcondition: `e == E(s)`.
Complexity: same as `E(s)`.

```
template<SeedSequence Sseq> void seed(E& e, Sseq& q);
```

Postcondition: `e == E(q)` (using the original value of `q`).
Complexity: same as `E(q)`.

```
void discard(E& e, unsigned long long z);
```

Effects: Advances the engine's state from $\mathtt{e}_i$ to $\mathtt{e}_{i+\mathtt{z}}$ by any means equivalent to the default implementation specified above.
Complexity: no worse than the complexity of `z` consecutive calls to `operator()`.
Note: 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 the default implementation specified above.

```
template<OutputStreamable OS> OS&& operator<<(OS&& os, const E& e);
```

Effects: 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 `e`'s current state. In the output, adjacent numbers are separated by one or more space characters.
Returns: the updated `os`.
Postcondition: The `os.`*fmtflags* and fill character are unchanged.

```
template<InputStreamable IS> IS&& operator>>(IS&& is, E& e);
```

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`.
Effects: With `is.fmtflags` set to `ios_base::dec`, sets `e`'s state as determined by reading its textual representation from `is`. If bad input is encountered, ensures that `e`'s state is unchanged by the operation and calls `is.setstate(ios::failbit)` (which may throw `ios::failure` [27.5.4.3]). If a textual representation written via `os << x` was subsequently read via `is >> v`, then `x == v` provided that there have been no intervening invocations of `x` or of `v`.
Returns: the updated `is`.
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

Expression	Return type	Pre/post-condition	Complexity
`E()`	—	Creates an engine with the same initial state as all other default-constructed engines of type `E`.	$\mathcal{O}$(size of state)
`E(x)`	—	Creates an engine that compares equal to `x`.	$\mathcal{O}$(size of state)
`E(s)`	—	Creates an engine with initial state determined by `s`.	$\mathcal{O}$(size of state)
`E(q)`[270]	—	Creates an engine with an initial state that depends on a sequence produced by one call to `q.generate`.	same as complexity of `q.generate` called on a sequence whose length is size of state
`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 $\mathtt{e}_i$ to $\mathtt{e}_{i+1}$ = TA($\mathtt{e}_i$) and returns GA($\mathtt{e}_i$).	per Table 110
`e.discard(z)` [271]	`void`	Advances `e`'s state $\mathtt{e}_i$ to $\mathtt{e}_{i+\mathtt{z}}$ 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`.	$\mathcal{O}$(size of state)
`x != y`	`bool`	`!(x == y)`.	$\mathcal{O}$(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.	$\mathcal{O}$(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()`.

Expression	Return type	Pre/post-condition	Complexity
`is >> v`	reference to the type of `is`	With `is.fmtflags` set to `ios_base::dec`, sets `v`'s state as determined by reading its textual representation from `is`. If bad input is encountered, ensures that `v`'s state is unchanged by the operation and calls `is.setstate(ios::failbit)` (which may throw `ios::failure` [27.5.4.3]). If a textual representation written via `os << x` was subsequently read via `is >> v`, then `x == v` provided that there have been no intervening invocations of `x` or of `v`. pre: `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 type's template specialization arguments `charT` and `traits` were respectively the same as those of `is`. post: The `is`.*fmtflags* are unchanged.	$\mathcal{O}$(size of state)

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 $\mathcal{O}$(size of state).

26.5.1.5 Concept `RandomNumberEngineAdaptor`Random number engine adaptor requirements [rand.req.adapt]

1 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. ~~Engines~~An engine `b` of type `B` adapted in this way ~~are~~is termed a *base engine~~s~~* 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.~~

2 ~~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^{th} of `A`'s base engines, $1 \leq i \leq n$, and `B`$_i$ denotes the type of `b`$_i$.~~

```
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&&...>;
}
```

3 The requirements of a random number engine type shall be interpreted as follows with respect to a random number engine adaptor type.

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

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

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

5 *Returns:* `true` if ~~each pair of corresponding b_i~~`a1`'s base engine is equal to `a2`'s base engine ~~are equal~~. Otherwise returns `false`.

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

6 *Effects:* ~~Each b_i~~The base engine is initialized~~, in turn,~~ with ~~the next available value from the list $s + 0, s + 1, \ldots$~~ `s`.

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

7 *Effects:* ~~Each b_i~~The base engine is initialized~~, in turn,~~ with `q` ~~as argument~~.

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

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

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

9 *Effects:* ~~For each b_i, in turn, invokes `RandomNumberEngine::seed(`b_i`, s)` with the next available value from the list $s + 0, s + 1, \ldots$.~~ With `b` as the base engine, invokes `b.seed(s)`.

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

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

11 `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 engine~~s~~. The size of `A`'s state shall be no less than the ~~sum of the base engines' respective sizes~~size of the base engine.

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

d) The textual representation of `A` shall include~~, in turn,~~ the textual representation of ~~each of~~ its base engine~~s~~.

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, $1 \leq i \leq n$, is B_i, and such that all remaining parameters, if any, have default values. The constructor shall create an engine adaptor initializing each b_i, in turn, with a copy of the value of the corresponding argument.~~

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, $p(z \mid a, b)$ or $P(z_i \mid a, b)$, to name specific parameters, or by writing, for example, $p(z \mid \{\texttt{p}\})$ or $P(z_i \mid \{\texttt{p}\})$, to denote a distribution's parameters `p` taken as a whole.

```
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 `typedef`s 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.

`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`.

```
void reset(D& d);
```

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

```
template<UniformRandomNumberGenerator URNG> result_type operator()(D& d, URNG& g);
```

10 *Effects:* With `p = param()`, the sequence of numbers returned by successive invocations with the same object `u` is randomly distributed according to the associated probability function $p(z \mid \{\mathtt{p}\})$ or $P(z_i \mid \{\mathtt{p}\})$.

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 `os << x` or of any `const` member function of `D` between any of the invocations `x(u)`.
b) If a textual representation is written using `os << x` and that representation is restored into the same or a different object `y` using `is >> y`, repeated invocations of `y(u)` shall produce the same sequence of numbers as would repeated invocations of `x(u)`.

11 *Complexity:* amortized constant number of invocations of `u`.

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

12 *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 $p(z \mid \{\mathtt{p}\})$ or $P(z_i \mid \{\mathtt{p}\})$.

```
param_type param(const D& d);
```

13 *Returns:* a value `p` such that `param(D(p)) == p`.
14 *Complexity:* no worse than the complexity of `D(p)`.

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

15 *Postcondition:* `param(d) == p`.
16 *Complexity:* no worse than the complexity of `D(p)`.

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

17 *Returns:* the greatest lower bound on the values potentially returned by `operator()`, as determined by the current values of the distribution's parameters.
18 *Complexity:* constant.

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

19 *Returns:* the least upper bound on the values potentially returned by `operator()`, as determined by the current values of the distribution's parameters.
20 *Complexity:* constant.

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

21 *Effects:* Writes to `os` a textual representation for the parameters and the additional internal data of `d`.
22 *Returns:* the updated `os`.
23 *Postcondition:* The `os.`*fmtflags* and fill character are unchanged.

```
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 satisfing 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

Expression	Return type	Pre/post-condition	Complexity
`D::result_type`	`T`	`T` is an arithmetic type (3.9.1).	compile-time
`D::param_type`	`P`	—	compile-time
`D()`	—	Creates a distribution whose behavior is indistinguishable from that of any other newly default-constructed distribution of type `D`.	constant
`D(p)`	—	Creates a distribution whose behavior is indistinguishable from that of a distribution newly constructed directly from the values used to construct `p`.	same as `p`'s construction
`d.reset()`	`void`	Subsequent uses of `d` do not depend on values produced by any engine prior to invoking `reset`.	constant
`x.param()`	`P`	Returns a value `p` such that `D(p).param() == p`.	no worse than the complexity of `D(p)`
`d.param(p)`	`void`	post: `d.param() == p`.	no worse than the complexity of `D(p)`

Expression	Return type	Pre/post-condition	Complexity
`d(g)`	`T`	With `p = d.param()`, the sequence of numbers returned by successive invocations with the same object `g` is randomly distributed according to the associated $p(z \mid \{\texttt{p}\})$ or $P(z_i \mid \{\texttt{p}\})$ function.	amortized constant number of invocations of `g`
`d(g,p)`	`T`	The sequence of numbers returned by successive invocations with the same objects `g` and `p` is randomly distributed according to the associated $p(z \mid \{\texttt{p}\})$ or $P(z_i \mid \{\texttt{p}\})$ function.	amortized constant number of invocations of `g`
`x.min()`	`T`	Returns `glb`.	constant
`x.max()`	`T`	Returns `lub`.	constant
`x == y`	`bool`	This operator is an equivalence relation. Returns `true` if `x.param() == y.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(g1)` and `y(g2)` whenever `g1 == g2`. Otherwise returns `false`.	constant
`x != y`	`bool`	`!(x == y)`.	same as `x == y`.
`os << x`	reference to the type of `os`	Writes to `os` a textual representation for the parameters and the additional internal data of `x`. post: The `os.`*fmtflags* and fill character are unchanged.	—

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]). pre: `is` provides a textual representation that was previously written using an `os` whose imbued locale and whose type's template specialization arguments `charT` and `traits` were the same as those of `is`. post: The `is.`*fmtflags* are unchanged.	—

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 `typedef`s for convenience of exposition only.

33 `P` shall satisfy the requirements of CopyConstructible (Table 34), CopyAssignable (Table 36), and EqualityComparable (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

```
typedef  D  distribution_type;
```

26.5.2 Header <random> synopsis [rand.synopsis]

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

```
namespace std {
  #include <initializer_list>

  // 26.5.1.2 ~~Concept SeedSequence~~
  ~~concept SeedSequence<typename S>~~

  // 26.5.1.3 ~~Concept UniformRandomNumberGenerator~~
  ~~concept UniformRandomNumberGenerator<typename U>~~
```

```
// 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<1u <= 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>~~
  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<0u < s && s < r && 0 < w && w <= numeric_limits<UIntType>::digits>~~
  class subtract_with_carry_engine;

// 26.5.4.1 Class template discard_block_engine
template<~~RandomNumberEngine~~class Engine, size_t p, size_t r>
  ~~requires True<1 <= r && r <= p>~~
  class discard_block_engine;

// 26.5.4.2 Class template independent_bits_engine
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;

// 26.5.4.3 Class template shuffle_order_engine
template<~~RandomNumberEngine~~class Engine, size_t k>
  ~~requires True<1u <= 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;
```

```
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;
```

```
  // 26.5.8.5.3 Class template piecewise_linear_distribution
  template<FloatingPointLikeclass RealType = double>
    requires FloatingPointType<RealType>
    class piecewise_linear_distribution;

} // namespace std
```

26.5.3 Random number engine class templates [rand.eng]

1 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.

2 Except where specified otherwise, the complexity of alleach functions specified in the following sectionsthis section 26.5.3 is constant.

3 Except where specified otherwise, no function described in this section 26.5.3 throws an exception.

4 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, Ddeclarations 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 26.5.3 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` [rand.eng.lcong]

1 A `linear_congruential_engine` random number engine produces unsigned integer random numbers. The state $\mathtt{x}_i$ 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 $\mathsf{TA}(\mathtt{x}_i) = (a \cdot \mathtt{x}_i + c) \bmod m$; the generation algorithm is $\mathsf{GA}(\mathtt{x}_i) = \mathtt{x}_{i+1}$.

```
template<UnsignedIntegralLikeclass UIntType, UIntType a, UIntType c, UIntType m>
 requires IntegralType<UIntType>
       && True<m == 0u || (a < m && c < m)>
 class linear_congruential_engine
{
public:
 // types
 typedef UIntType result_type;

 // engine characteristics
 static constexpr result_type multiplier = a;
 static constexpr result_type increment = c;
 static constexpr result_type modulus = m;
 static constexpr result_type min() { return c == 0u ? 1u: 0u };
 static constexpr result_type max() { return m - 1u };
 static constexpr result_type default_seed = 1u;

 // constructors and seeding functions
 explicit linear_congruential_engine(result_type s = default_seed);
 template<SeedSequenceclass Sseq> explicit linear_congruential_engine(Sseq& q);
 void seed(result_type s = default_seed);
 template<SeedSequenceclass Sseq> void seed(Sseq& q);

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

```
};
```

2 If the template parameter `m` is 0, the modulus m used throughout this section 26.5.3.1 is `numeric_limits<result_type>::max()` plus 1. [*Note:* m need not be representable as a value of type `result_type`. — *end note*]

3 If the template parameter `m` is not 0, the following relations shall hold: `a < m` and `c < m`.

4 The textual representation consists of the value of $\mathtt{x}_i$.

```
explicit linear_congruential_engine(result_type s = default_seed);
```

5 *Effects:* Constructs a `linear_congruential_engine` object. If $c \bmod m$ 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.

```
template<SeedSequenceclass Sseq> explicit linear_congruential_engine(Sseq& q);
```

6 *Effects:* Constructs a `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 `q.generate(`$a+0$`,` $a+k+3$`)` and then computes $S = \left(\sum_{j=0}^{k-1} a_{j+3} \cdot 2^{32j}\right) \bmod m$. If $c \bmod m$ 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 `mersenne_twister_engine` [rand.eng.mers]

1 A `mersenne_twister_engine` random number engine[272] produces unsigned integer random numbers in the closed interval $[0, 2^w - 1]$. The state $\mathtt{x}_i$ of a `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.

2 The transition algorithm employs a twisted generalized feedback shift register defined by shift values n 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 *tempered* (*i.e.*, scrambled) according to a bit-scrambling matrix defined by values u, d, s, b, t, c, and ℓ.

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

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 $z_1 = X_i \mathsf{\ xor\ } \big((X_i \mathsf{\ rshift\ } u) \mathsf{\ bitand\ } d\big)$.

b) Let $z_2 = z_1 \mathsf{\ xor\ } \big((z_1 \mathsf{\ lshift}_w\ s) \mathsf{\ bitand\ } b\big)$.

c) Let $z_3 = z_2 \mathsf{\ xor\ } \big((z_2 \mathsf{\ lshift}_w\ t) \mathsf{\ bitand\ } c\big)$.

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

```
template<UnsignedIntegralLikeclass 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<1u <= 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
           && d <= (1u<<w) - 1u && f <= (1u<<w) - 1u>
 class mersenne_twister_engine
```

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.

```
{
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_b = b;
 static constexpr size_t tempering_t = t;
 static constexpr UIntType tempering_c = c;
 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  2^w − 1; }
 static constexpr result_type default_seed = 5489u;

 // constructors and seeding functions
 explicit mersenne_twister_engine(result_type value = default_seed);
 template<SeedSequenceclass Sseq> explicit mersenne_twister_engine(Sseq& q);
 void seed(result_type value = default_seed);
 template<SeedSequenceclass 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`, `d <= (1u«w) - 1u`, and `f <= (1u«w) - 1u`.

5 The textual representation of $\mathtt{x}_i$ consists of the values of $X_{i-n}, \ldots, X_{i-1}$, in that order.

`explicit mersenne_twister_engine(result_type value = default_seed);`

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

$$\left[f \cdot \left(X_{i-1} \mathsf{xor} \left(X_{i-1} \mathsf{rshift} \left(w-2\right)\right)\right) + i \bmod n\right] \bmod 2^w \ .$$

7 *Complexity:* $\mathcal{O}(n)$.

`template<SeedSequenceclass Sseq> explicit mersenne_twister_engine(Sseq& q);`

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 `q.generate(`$a + 0$`,` $a + n \cdot k$`)` and then, iteratively for $i = -n, \ldots, -1$, sets X_i to $\left(\sum_{j=0}^{k-1} a_{k(i+n)+j} \cdot 2^{32j}\right) \bmod 2^w$. 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}.

26.5.3.3 Class template `subtract_with_carry_engine` [rand.eng.sub]

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

2 The state $\mathtt{x}_i$ of a `subtract_with_carry_engine` object `x` is of size $\mathcal{O}(r)$, 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 $\mathtt{x}_i$ additionally consists of an integer c (known as the *carry*) whose value is either 0 or 1.

3 The state transition is performed as follows:

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

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

[*Note:* This algorithm corresponds to a modular linear function of the form $\mathsf{TA}(\mathtt{x}_i) = (a \cdot \mathtt{x}_i) \bmod b$, where b is of the form $m^r - m^s + 1$ and $a = b - (b-1)/m$. — *end note*]

4 The generation algorithm is given by $\mathsf{GA}(\mathtt{x}_i) = y$, where y is the value produced as a result of advancing the engine's state as described above.

```
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<~~SeedSequence~~class Sseq> explicit subtract_with_carry_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);
};
```

5 The following relations shall hold: `0u < s`, `s < r`, `0 < w`, and `w <= numeric_limits<UIntType>::digits`.

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

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

7 *Effects:* Constructs a `subtract_with_carry_engine` object. Sets the values of $X_{-r}, \ldots, X_{-1}$, in that order, as specified below. If X_{-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:

```
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 $\left(\sum_{j=0}^{n-1} z_j \cdot 2^{32j}\right) \bmod m$.

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

```
template<SeedSequenceclass Sseq> explicit subtract_with_carry_engine(Sseq& q);
```

9 *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(`$a+0$`,` $a+r \cdot k$`)` and then, iteratively for $i = -r, \ldots, -1$, sets X_i to $\left(\sum_{j=0}^{k-1} a_{k(i+r)+j} \cdot 2^{32j}\right) \bmod m$. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.

26.5.4 Random number engine adaptor class templates [rand.adapt]

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 alleach functions specified in the following sectionsthis 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 hereein 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, Ddeclarations 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` [rand.adapt.disc]

1 A `discard_block_engine` random number engine adaptor produces random numbers selected from those produced by some base engine e. The state $\mathtt{x}_i$ of a `discard_block_engine` engine adaptor object `x` consists of the state $\mathtt{e}_i$ 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 $\mathtt{e}_i$ to $\mathtt{e}_{i+p-r}$ and set n to 0. In any case, then increment n and advance `e`'s then-current state $\mathtt{e}_j$ to $\mathtt{e}_{j+1}$.

3 The generation algorithm yields the value returned by the last invocation of `e()` while advancing `e`'s state as described above.

```
template<RandomNumberEngineclass Engine, size_t p, size_t r>
 requires True<1 <= r && r <= 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<SeedSequenceclass Sseq> explicit discard_block_engine(Sseq& q);
  void seed();
  void seed(result_type s);
  template<SeedSequenceclass 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` [rand.adapt.ibits]

1 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 $\mathtt{x}_i$ of an `independent_bits_engine` engine adaptor object `x` consists of the state $\mathtt{e}_i$ 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 $R = \mathtt{e.max()} - \mathtt{e.min()} + 1$ and $m = \lfloor \log_2 R \rfloor$.

b) With n as determined below, let $w_0 = \lfloor w/n \rfloor$, $n_0 = n - w \bmod n$, $y_0 = 2^{w_0} \lfloor R/2^{w_0} \rfloor$, and $y_1 = 2^{w_0+1} \lfloor R/2^{w_0+1} \rfloor$.

c) Let $n = \lceil w/m \rceil$ if and only if the relation $R - y_0 \le \lfloor y_0/n \rfloor$ holds as a result. Otherwise let $n = 1 + \lceil w/m \rceil$.

[*Note:* The relation $w = n_0 w_0 + (n - n_0)(w_0 + 1)$ always holds. — *end note*]

3 The transition algorithm is carried out by invoking `e()` as often as needed to obtain n_0 values less than $y_0 + \mathtt{e.min()}$ and $n - n_0$ values less than $y_1 + \mathtt{e.min()}$.

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;
for (k = 0; k ≠ n0; k += 1) {
 do u = e() - e.min(); while (u ≥ y0);
 S = 2^w0 · S + u mod 2^w0;
}
for (k = n0; k ≠ n; k += 1) {
 do u = e() - e.min(); while (u ≥ y1);
 S = 2^(w0+1) · S + u mod 2^(w0+1);
}
```

```
template<RandomNumberEngineclass Engine, size_t w, UnsignedIntegralLikeclass 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<SeedSequenceclass Sseq> explicit independent_bits_engine(Sseq& q);
 void seed();
 void seed(result_type s);
 template<SeedSequenceclass 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 $\mathtt{x}_i$ of a `shuffle_order_engine` engine adaptor object `x` consists of the state $\mathtt{e}_i$ 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<RandomNumberEngineclass Engine, size_t k>
 requires True<1u <= k>
 class shuffle_order_engine
{
public:
 // types
 typedef typename Engine::result_type result_type;

 // engine characteristics
 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<SeedSequenceclass Sseq> explicit shuffle_order_engine(Sseq& q);
  void seed();
  void seed(result_type s);
  template<SeedSequenceclass 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;
```

1 *Required behavior:* The 10000^{th} 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;
```

2 *Required behavior:* The 10000^{th} 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;
```

3 *Required behavior:* The 10000^{th} 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,
      0x5555555555555555,17,
```

```
    0x71d67fffeda60000,37,
    0xfff7eee000000000,43,
    6364136223846793005>
    mt19937_64;
```

4 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `mt19937_64` shall produce the value 9981545732273789042.

```
typedef subtract_with_carry_engine<uint_fast32_t, 24, 10, 24>
    ranlux24_base;
```

5 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `ranlux24_base` shall produce the value 7937952.

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

6 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `ranlux48_base` shall produce the value 61839128582725.

```
typedef discard_block_engine<ranlux24_base, 223, 23>
    ranlux24;
```

7 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `ranlux24` shall produce the value 9901578.

```
typedef discard_block_engine<ranlux48_base, 389, 11>
    ranlux48
```

8 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `ranlux48` shall produce the value 249142670248501.

```
typedef shuffle_order_engine<minstd_rand0,256>
    knuth_b;
```

9 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `knuth_b` shall produce the value 1112339016.

```
typedef implementation-defined
    default_random_engine;
```

10 *Required behavior:* A concept map `RandomNumberEngine<default_random_engine>`, or equivalent, shall be defined in namespace `std` so as to provide mappings from free functions to the corresponding member functions.

11 *Remark:* The choice of engine type named by this `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 `typedef` need not generate identical sequences across implementations. — *end note*]

26.5.6 Class `random_device` [rand.device]

1 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.~~

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

3 [Editor's note: This paragraph has been inserted to address LWG Issue 1068. However, is is unclear why this type should be made to meet a requirement not imposed on other URNG types. Further, because of possible performance implications, it is unclear whether this is a good idea at all.] In addition to the functionality shown in the following synopsis, class `random_device` shall meet the requirements of a MoveConstructible (Table 33) and a MoveAssignable (Table 35) type.

```
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;
};
```

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

4 *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]

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

`double entropy() const;`

6 *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 $\log_2(\texttt{max}() + 1)$.

7 *Throws:* Nothing.

`result_type operator()();`

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

9 *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 [rand.util]

26.5.7.1 Class `seed_seq` [rand.util.seedseq]

1 No function described in this section 26.5.7.1 throws an exception.

2 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.

3 [Editor's note: This paragraph has been inserted to address LWG Issue 1069. However, it is unclear (a) whether this is a good idea, and (b) why this type should be made to meet a requirement not imposed on other Seed Sequence types.] In addition to the functionality shown in the following synopsis, class `seed_seq` shall meet the requirements of a MoveConstructible (Table 33) and a MoveAssignable (Table 35) type.

```
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~~<auto,const result_type&> Iter~~>
   void param(~~Iter~~OutputIterator dest) const;

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

`seed_seq();`

4 *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);
```

5 *Requires:* `T` shall be an integer type.

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

`template<class InputIterator ~~Iter~~>`

```
  requires IntegralLike<Iter::value_type>
        && IntegralType<Iter::value_type>
  seed_seq(IterInputIterator begin, IterInputIterator end);
```

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

8 *Effects:* Constructs a `seed_seq` object by the following algorithm:

```
for( InputIterator s = begin; s != end; ++s)
 v.push_back((*s) mod 2^32);
```

```
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(IterRandomAccessIterator begin, IterRandomAccessIterator end);
```

9 *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.

10 *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 xor (x 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

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

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

$$\begin{aligned} r_1 &= 1664525 \cdot \mathtt{T}\,(\mathtt{begin}[k]\ \mathsf{xor}\ \mathtt{begin}[k+p]\ \mathsf{xor}\ \mathtt{begin}[k-1]) \\ r_2 &= r_1 + \begin{cases} s & , k = 0 \\ k \bmod n + \mathtt{v}[k-1] & , 0 < k \leq s \\ k \bmod n & , s < k \end{cases} \end{aligned}$$

and, in order, increment `begin`$[k+p]$ by r_1, increment `begin`$[x+q]$ by r_2, and set `begin[k]` 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

$$\begin{aligned} r_3 &= 1566083941 \cdot \mathtt{T}\,(\mathtt{begin}[k] + \mathtt{begin}[k+p] + \mathtt{begin}[k-1]) \\ r_4 &= r_3 - (k \bmod n) \end{aligned}$$

and, in order, update `begin`$[k+p]$ by xoring it with r_4, update `begin`$[k+q]$ by xoring it with r_3, and set `begin`$[k]$ to r_4.

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

11 *Returns:* The number of 32-bit units that would be returned by a call to `param()`.

12 *Complexity:* constant time.

```
template<class OutputIterator<auto,const result_type&> Iter>
  void param(IterOutputIterator dest) const;
```

13 *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`.

14 *Effects:* Copies the sequence of prepared 32-bit units to the given destination, as if by executing the following statement:

```
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, $0 \le t_j < 1$, 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<FloatingPointLikeclass RealType, size_t bits, UniformRandomNumberGeneratorclass URNG>
 requires FloatingPointType<RealType>
 RealType generate_canonical(URNG& g);
```

3 *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.max() - g.min() + 1`.

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

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

using arithmetic of type `RealType`.

5 *Returns:* S/R^k.

6 *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` so as to provide mappings from free functions to the corresponding member functions. Descriptions are provided hereinin 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, Ddeclarations 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, $a \le i \le b$, distributed according to the constant discrete probability function

$$P(i \mid a, b) = 1/(b - a + 1) \ .$$

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

```
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]

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

$$p(x \mid a, b) = 1/(b - a) \ .$$

```
template<~~FloatingPointLike~~class RealType = double>
 ~~requires FloatingPointType<RealType>~~
 class uniform_real_distribution
{
public:
 // types
```

```
  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<~~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_real_distribution(RealType a = 0.0, RealType b = 1.0);`

2 *Requires:* `a` $\leq$ `b` and `b` $-$ `a` $\leq$ `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

$$P(b \mid p) = \begin{cases} p & \text{if} \quad b = \texttt{true} \\ 1 - p & \text{if} \quad b = \texttt{false} \end{cases} .$$

```
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;
};
```

```
explicit bernoulli_distribution(double p = 0.5);
```

2 *Requires:* $0 \le \mathtt{p} \le 1$.

3 *Effects:* Constructs a `bernoulli_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.

26.5.8.2.2 Class template `binomial_distribution` [rand.dist.bern.bin]

1 A `binomial_distribution` random number distribution produces integer values $i \ge 0$ distributed according to the discrete probability function

$$P(i \mid t, p) = \binom{t}{i} \cdot p^i \cdot (1-p)^{t-i} .$$

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

 // constructors and reset functions
 explicit binomial_distribution(IntType t = 1, double p = 0.5);
 explicit 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 t() const;
  double p() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
 };
```

```
explicit binomial_distribution(IntType t = 1, double p = 0.5);
```

2 *Requires:* $0 \le$ p ≤ 1 and $0 \le$ t.

3 *Effects:* Constructs a `binomial_distribution` object; `t` and `p` correspond to the respective parameters of the distribution.

```
IntType t() const;
```

4 *Returns:* The value of the `t` parameter with which the object was constructed.

```
double p() const;
```

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

26.5.8.2.3 Class template `geometric_distribution` [rand.dist.bern.geo]

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

$$P(i \mid p) = p \cdot (1-p)^i \ .$$

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

 // constructors and reset functions
 explicit geometric_distribution(double p = 0.5);
 explicit geometric_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;
 };
```

```
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

$$P(i \mid k, p) = \binom{k+i-1}{i} \cdot p^k \cdot (1-p)^i \ .$$

```
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:* $0 < p \leq 1$ 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.

```
double p() const;
```

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

26.5.8.3 Poisson distributions [rand.dist.pois]

26.5.8.3.1 Class template `poisson_distribution` [rand.dist.pois.poisson]

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

$$P(i \mid \mu) = \frac{e^{-\mu}\mu^i}{i\,!} \ .$$

The distribution parameter μ is also known as this distribution's *mean* .

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

 // constructors and reset functions
 explicit poisson_distribution(double mean = 1.0);
 explicit poisson_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 mean() const;
 param_type param() const;
 void param(const param_type& parm);
 result_type min() const;
 result_type max() const;
};
```

```
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` [rand.dist.pois.exp]

An `exponential_distribution` random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x \mid \lambda) = \lambda e^{-\lambda x} \ .$$

```
template<FloatingPointLikeclass 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<UniformRandomNumberGeneratorclass URNG>
   result_type operator()(URNG& g);
 template<UniformRandomNumberGeneratorclass 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` [rand.dist.pois.gamma]

1 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<FloatingPointLikeclass 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);
```

2 *Requires:* $0 <$ `alpha` and $0 <$ `beta`.

3 *Effects:* Constructs a `gamma_distribution` object; `alpha` and `beta` correspond to the parameters of the distribution.

```
RealType alpha() const;
```

4 *Returns:* The value of the `alpha` parameter with which the object was constructed.

```
RealType beta() const;
```

5 *Returns:* The value of the `beta` parameter with which the object was constructed.

26.5.8.3.4 Class template `weibull_distribution` [rand.dist.pois.weibull]

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

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

```
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);
```

2 *Requires:* $0 <$ `a` and $0 <$ `b`.

3 *Effects:* Constructs a `weibull_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

```
RealType a() const;
```

4 *Returns:* The value of the `a` parameter with which the object was constructed.

```
RealType b() const;
```

5 *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]

1 An `extreme_value_distribution` random number distribution produces random numbers x distributed according to the probability density function[276]

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

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

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

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

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);
```

2 *Requires:* $0 < \texttt{b}$.

3 *Effects:* Constructs an `extreme_value_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

```
RealType a() const;
```

4 *Returns:* The value of the `a` parameter with which the object was constructed.

```
RealType b() const;
```

5 *Returns:* The value of the `b` parameter with which the object was constructed.

26.5.8.4 Normal distributions [rand.dist.norm]

26.5.8.4.1 Class template `normal_distribution` [rand.dist.norm.normal]

1 A `normal_distribution` random number distribution produces random numbers x distributed according to the probability density function

$$p(x \mid \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

The distribution parameters μ and σ 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);
```

```
  // 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);
```

2 *Requires:* $0 <$ `stddev`.

3 *Effects:* Constructs a `normal_distribution` object; `mean` and `stddev` correspond to the respective parameters of the distribution.

```
RealType mean() const;
```

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

```
RealType stddev() const;
```

5 *Returns:* The value of the `stddev` parameter with which the object was constructed.

26.5.8.4.2 Class template `lognormal_distribution` [rand.dist.norm.lognormal]

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

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

```
template<FloatingPointLikeclass 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<UniformRandomNumberGeneratorclass URNG>
   result_type operator()(URNG& g);
 template<UniformRandomNumberGeneratorclass URNG>
   result_type operator()(URNG& g, const param_type& parm);

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

```
  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);
```

2 *Requires:* $0 < \texttt{s}$.

3 *Effects:* Constructs a `lognormal_distribution` object; `m` and `s` correspond to the respective parameters of the distribution.

```
RealType m() const;
```

4 *Returns:* The value of the `m` parameter with which the object was constructed.

```
RealType s() const;
```

5 *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]

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

$$p(x \mid n) = \frac{x^{(n/2)-1} \cdot e^{-x/2}}{\Gamma(n/2) \cdot 2^{n/2}} \ .$$

```
 template<FloatingPointLikeclass RealType = double>
  requires FloatingPointType<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<UniformRandomNumberGeneratorclass URNG>
    result_type operator()(URNG& g);
  template<UniformRandomNumberGeneratorclass 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 *Requires:* $0 < \texttt{n}$.
3 *Effects:* Constructs a `chi_squared_distribution` object; `n` corresponds to the parameter of the distribution.

```
RealType n() const;
```

4 *Returns:* The value of the `n` parameter with which the object was constructed.

26.5.8.4.4 Class template `cauchy_distribution` [rand.dist.norm.cauchy]

1 A `cauchy_distribution` random number distribution produces random numbers x distributed according to the probability density function

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

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

 // constructor and reset functions
 explicit cauchy_distribution(RealType a = 0.0, RealType b = 1.0);
 explicit cauchy_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 cauchy_distribution(RealType a = 0.0, RealType b = 1.0);
```

2 *Requires:* $0 < \texttt{b}$.
3 *Effects:* Constructs a `cauchy_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

```
RealType a() const;
```

4 *Returns:* The value of the `a` parameter with which the object was constructed.

```
RealType b() const;
```

5 *Returns:* The value of the `b` parameter with which the object was constructed.

26.5.8.4.5 Class template `fisher_f_distribution` [rand.dist.norm.f]

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

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

```
template<FloatingPointLikeclass 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<UniformRandomNumberGeneratorclass URNG>
   result_type operator()(URNG& g);
 template<UniformRandomNumberGeneratorclass 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);
```

2 *Requires:* $0 <$ `m` and $0 <$ `n`.

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

```
RealType m() const;
```

4 *Returns:* The value of the `m` parameter with which the object was constructed.

```
RealType n() const;
```

5 *Returns:* The value of the `n` parameter with which the object was constructed.

26.5.8.4.6 Class template `student_t_distribution` [rand.dist.norm.t]

1 A `student_t_distribution` random number distribution produces random numbers x distributed according to the probability density function

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

```
template<FloatingPointLikeclass RealType = double>
 requires 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<UniformRandomNumberGeneratorclass URNG>
   result_type operator()(URNG& g);
 template<UniformRandomNumberGeneratorclass 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 student_t_distribution(RealType n = 1);
```

2 *Requires:* $0 < \mathtt{n}$.

3 *Effects:* Constructs a `student_t_distribution` object; `n` corresponds to the parameter of the distribution.

```
RealType n() const;
```

4 *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 `discrete_distribution` [rand.dist.samp.discrete]

1 A `discrete_distribution` random number distribution produces random integers i, $0 \le i < n$, distributed according to the discrete probability function

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

2 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 *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(~~Iter~~InputIterator firstW, ~~Iter~~InputIterator lastW);
 discrete_distribution(initializer_list<double> wl);
 template<~~Callable<auto, double> Func~~class UnaryOperation>
   ~~requires Convertible<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();`

3 *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(~~Iter~~InputIterator firstW, ~~Iter~~InputIterator lastW);
```

4 *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 `firstW == lastW`, let $n = 1$ and $w_0 = 1$. Otherwise, $[$`firstW`, `lastW`$)$ shall form a sequence w of length $n > 0$.

5 *Effects:* Constructs a `discrete_distribution` object with probabilities given by the formula above.

`discrete_distribution(initializer_list<double> wl);`

6 *Effects:* Same as `discrete_distribution(wl.begin(), wl.end())`.

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

```
requires Convertible<Func::result_type, double>
discrete_distribution(size_t nw, double xmin, double xmax, FuncUnaryOperation fw);
```

7 *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.

8 *Effects:* Constructs a `discrete_distribution` object with probabilities given by the formula above, using the following values: If `nw` $= 0$, let $w_0 = 1$. Otherwise, let $w_k =$ `fw`(`xmin` $+ k \cdot \delta + \delta/2$) for $k = 0, \ldots, n-1$.

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

```
vector<double> probabilities() const;
```

10 *Returns:* A `vector<double>` whose `size` member returns n and whose `operator[]` member returns p_k when invoked with argument k for $k = 0, \ldots, n-1$.

26.5.8.5.2 Class template `piecewise_constant_distribution` [rand.dist.samp.pconst]

1 A `piecewise_constant_distribution` random number distribution produces random numbers x, $b_0 \le x < b_n$, uniformly distributed over each subinterval $[b_i, b_{i+1})$ according to the probability density function

$$p(x \mid b_0, \ldots, b_n,\ \rho_0, \ldots, \rho_{n-1}) = \rho_i \text{ , for } b_i \le x < b_{i+1} \text{ .}$$

2 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 distribution parameters are calculated as:

$$\rho_k = \frac{w_k}{S \cdot (b_{k+1} - b_k)} \text{ for } k = 0, \ldots, n-1,$$

in which the values w_k, commonly known as the *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<FloatingPointLikeclass 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 IterWclass InputIteratorB, class InputIteratorW>
   requires Convertible<IterB::value_type, result_type> && Convertible<IterW::value_type, double>
   piecewise_constant_distribution(IterBInputIteratorB firstB, IterBInputIteratorB lastB,
                                   IterWInputIteratorW firstW);
 template<Callable<auto, RealType> Funcclass UnaryOperation>
   requires Convertible<Func::result_type, double>
   piecewise_constant_distribution(initializer_list<RealType> bl, FuncUnaryOperation fw);
 template<Callable<auto, double> Funcclass UnaryOperation>
   requires Convertible<Func::result_type, double>
   piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax, FuncUnaryOperation fw);
 explicit piecewise_constant_distribution(const param_type& parm);
 void reset();
```

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

  // property functions
  vector<RealTyperesult_type> 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();
```

3 *Effects:* Constructs a `piecewise_constant_distribution` object with $n = 1$, $\rho_0 = 1$, $b_0 = 0$, and $b_1 = 1$.

```
template<InputIterator IterB, InputIterator IterWclass InputIteratorB, class InputIteratorW>
 requires Convertible<IterB::value_type, result_type> && Convertible<IterW::value_type, double>
 piecewise_constant_distribution(IterBInputIteratorB firstB, IterBInputIteratorB lastB,
                                 IterWInputIteratorW firstW);
```

4 *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, $\big[$`firstB`, `lastB`$\big)$ 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.

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

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

6 *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.

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

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

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

9 *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.

10 *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 $k = 0, \ldots, n–1$.

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

```
vector<result_type> intervals() const;
```

12 *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;
```

13 *Returns:* A `vector<result_type>` whose `size` member returns n and whose `operator[]` member returns ρ_k when invoked with argument k for $k = 0, \ldots, n-1$.

26.5.8.5.3 Class template `piecewise_linear_distribution` [rand.dist.samp.plinear]

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

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

2 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:

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

```
template<FloatingPointLikeclass 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 IterWclass InputIteratorB, class InputIteratorW>
   requires Convertible<IterB::value_type, result_type> && Convertible<IterW::value_type, double>
   piecewise_linear_distribution(IterBInputIteratorB firstB, IterBInputIteratorB lastB,
                                 IterWInputIteratorW firstW);
 template<Callable<auto, RealType> FuncclassUnaryOperation>
   requires Convertible<Func::result_type, double>
   piecewise_linear_distribution(initializer_list<RealType> bl, FuncUnaryOperation fw);
 template<Callable<auto, double> FuncclassUnaryOperation>
   requires Convertible<Func::result_type, double>
   piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, FuncUnaryOperation fw);
 explicit piecewise_linear_distribution(const param_type& parm);
 void reset();

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

```
  // property functions
  vector<RealTyperesult_type> 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();`

3 *Effects:* Constructs a `piecewise_linear_distribution` object with $n = 1$, $\rho_0 = \rho_1 = 1$, $b_0 = 0$, and $b_1 = 1$.

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

4 *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 $n = 1$, $\rho_0 = \rho_1 = 1$, $b_0 = 0$, and $b_1 = 1$. Otherwise, $\big[$`firstB`, `lastB`$\big)$ shall form a sequence b of length $n + 1$, the length of the sequence w starting from `firstW` shall be at least $n + 1$, and any w_k for $k \geq n + 1$ shall be ignored by the distribution.

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

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

6 *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.

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

8 *Complexity:* The number of invocations of `fw` shall not exceed $n + 1$.

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

9 *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.

10 *Effects:* Constructs a `piecewise_linear_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`$(b_k + \delta)$ for $k = 0, \ldots, n$.

11 *Complexity:* The number of invocations of `fw` shall not exceed $n + 1$.

`vector<result_type> intervals() const;`

12 *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;
```

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

26.6 Numeric arrays [numarray]

26.6.1 Header <valarray> synopsis [valarray.syn]

```
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 indirected 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&);
```