Random Number Generation in C++0X: A Comprehensive Proposal, version 3

Document number:	WG21/N2079= J16/06-0149
Date:	2006-09-08
Revises:	WG21/N2032 = J16/06-0102
Project:	Programming Language C++
Reference:	ISO/IEC IS 14882:2003(E)

Reply to: Walter E. Brown <`wb@fnal.gov`>
Mark Fischler <`mf@fnal.gov`>
Jim Kowalkowski <`jbk@fnal.gov`>
Marc Paterno <`paterno@fnal.gov`>
CEPA Dept., Computing Division
Fermi National Accelerator Laboratory
Batavia, Illinois 60510-0500
U.S.A.

This document revises N2032 = Brown, *et al.*: **Random Number Generation in C++0X: A Comprehensive Proposal, version 2**. It incorporates all known corrections to that paper's language and typography, and, with one important difference, subsumes the changes recommended in N2033 = Brown, *et al.*: **Proposal to Consolidate the Subtract-with-Carry Engines**.

Each local change in wording from the previous version of this paper is indicated as either added or ~~deleted~~ text. A subsection to be deleted in its entirety is so annotated at its start: *Delete the entirety of this subsection.* In addition, the title of such a subsection includes the annotation: [TO BE DELETED]. Changes in the index are not specially marked.

Of all the changes indicated in this document, we believe only one to be fundamental: We are herewith proposing that a URNG's `result_type` be an unsigned integral type This recommendation is not taken at all lightly; however the consequences of coping with real-valued generators and engines have consistently proven problematic, both in theory and in practice. We now recognize that real-valued engines are historical artifacts of the era in which real values often substituted for the absence of very long integral types. Given the recent improvement in C++0X support for such integral types (especially the `unsigned long long` type and the `uint*_t typedefs` provided in `<cstdint>`), we believe it is neither necessary nor useful to retain these artifacts any longer. We have updated this document accordingly, thereby achieving a number of important simplifications to the text as well as to our reference implementation. We are planning a companion paper, **Proposal to Retain Only Integral Engines**, that will provide some additional detail.

At the urging of several experts in the field, we have added a new type, `seed_seq`, as well as algorithms whereby engines can take advantage of this new type. These algorithms will take the place of the generator-based constructor and seeding algorithms that have been criticized in the past.

It has turned out to be far more difficult to correctly specify the behavior of the function template `generate_canonical` [26.4.7.2] than to write its code. The first attempt (in the predecessor document N2032) was correct for the integer case, but upon further reflection did not adequately address floating-point. We have now specified detailed algorithms, one for each case. Additionally, we have adjusted the interface in response to numerous requests for variations not limited to a single call of the supplied generator. As an alternative, we have provided a new engine adaptor (`random_bits_engine`) with a similar interface in order to provide two options for the Committee's consideration.

We would like to acknowledge the Fermi National Accelerator Laboratory's Computing Division, sponsors of our participation in the C++ standards effort, for its support. Thanks also to Paolo Carlini, Charles Karney, Bradley Lucier, Jens Maurer, Makoto Matsumoto, and Stephan Tolksdorf for their respective helpful correspondence regarding several technical aspects of the proposal.

Contents

List of Tables

26 Numerics library [lib.numerics]

26.4 Random number generation [lib.random.numbers]

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

2 ~~F~~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 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 [basic.fundamental] 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 `T`.

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.

26.4.1 Requirements [lib.rand.req]

26.4.1.1 General requirements [lib.rand.req.genl]

1 Throughout this subclause 26.4, the effect of instantiating a template

a) that has a template type parameter named `UniformRandomNumberGenerator` is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of uniform random number generator [26.4.1.2].

b) that has a template type parameter named `Engine` is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of ~~uniform~~ random number engine [26.4.1.3].

c) 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`.

d) 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`, ~~or~~ `unsigned long`, or `unsigned long long`.

e) 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`, ~~or~~ `unsigned long`, or `unsigned long long`.

2 All members declared `static const` in any of the following class templates shall be defined in such a way that they are usable as integral constant expressions.

26.4.1.2 Uniform random number generator requirements [lib.rand.req.urng]

1 A class `X` satisfies the requirements of a *uniform random number generator* if the expressions shown in table 1 are valid and have the indicated semantics. In that table,

a) `T` is the type named by `X`'s associated `result_type`, and

b) `u` is a value of `X`.

Table 1: Uniform random number generator requirements

expression	return type	pre/post-condition	complexity
`X::result_type`	`T`	`T` is an ~~arithmetic~~unsigned integer type [basic.fundamental] ~~other than `bool`~~.	compile-time
`u()`	`T`	~~If `X` is integral, r~~Returns a value in the closed interval [`X::min`, `X::max`]~~; otherwise, returns a value in the open interval (0,1)~~.	amortized constant
`X::min`	`T`~~, if `X` is integral; otherwise `int`.~~	~~If `X` is integral, d~~Denotes the least value potentially returned by `operator()`~~; otherwise denotes 0~~.	compile-time
`X::max`	`T`~~, if `X` is integral; otherwise `int`.~~	~~If `X` is integral, d~~Denotes the greatest value potentially returned by `operator()`~~; otherwise denotes 1~~.	compile-time

26.4.1.3 Random number engine requirements [lib.rand.req.eng]

1 A class `X` that ~~`X`~~ satisfies the requirements of a uniform random number generator [26.4.1.2] also satisfies the requirements of a *random number engine* if the expressions shown in table 2 are valid and have the indicated semantics, and if `X` also

satisfies all other requirements of this section 26.4.1.3. In that table and throughout this section 26.4.1.3,

a) `T` is the type named by `X`'s associated `result_type`;

b) ~~`t` is a value of `T`;~~

c) `u` is a value of `X`, `v` is an lvalue of `X`, `x` and `y` are (possibly `const`) values of `X`;

d) `s` is a value of ~~integral~~arithmetic type [basic.fundamental];

e) ~~`g` is an lvalue, of a type other than `X`, that defines a zero-argument function object returning values of type `unsigned long`~~ `q` is an lvalue of type `seed_seq` [26.4.7.1] ;

f) `z` is a value of type `size_t`;

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 [lib.strings] and [lib.input.output].

2 A random number engine object `x` has at any given time a state x_i for some integer $i \geq 0$. Upon construction, a random number engine `x` has an initial state x_0. An engine's state may be established by invoking a constructor, `seed` member function, `operator=`, or a suitable `operator>>`.

3 The specification of each random number engine defines the size of its state in multiples of the size of its `result_type`, given as an integral constant expression. The specification of each random number engine also defines

a) the *transition algorithm* TA by which the engine's state x_i is advanced to its *successor state* x_{i+1}, and

b) the *generation algorithm* GA by which an engine's state is mapped to a value of type `result_type`.

Table 2: Random number engine requirements

expression	return type	pre/post-condition	complexity
`X()`	—	Creates an engine with the same initial state as all other default-constructed engines of type `X`.	$\mathcal{O}$(size of state)
`X(x)`	—	Creates an engine that compares equal to `x`.	$\mathcal{O}$(size of state)
`X(s)`	—	Creates an engine with initial state determined by `static_cast<`~~`unsigned long`~~`X::result_type>(s)`.	$\mathcal{O}$(size of state)

expression	return type	pre/post-condition	complexity
`X(`~~`g`~~`q)`[1]	—	With $n =$ `q.size()`, ~~C~~creates an engine `u` with initial state determined ~~by the results of successive invocations of g~~as follows: If n is 0, `u == X()`; otherwise, the initial state depends on a sequence produced by one call to `q.randomize`. ~~Throws what and when g throws.~~	$\mathscr{O}$(max(n,size of state))
`u.seed()`	`void`	post: `u == X()`	same as `X()`
`u.seed(s)`	`void`	post: `u == X(s)`	same as `X(s)`
`u.seed(`~~`g`~~`q)`	`void`	post: ~~If g does not throw,~~ `u ==` ~~`v`, where the state of v is as if constructed by~~ `X(`~~`g`~~`q)`. ~~Otherwise, the exception is rethrown and the engine's state is deemed invalid. Thereafter, further use of u is undefined except for destruction or invoking a function that establishes a valid state.~~	same as `X(`~~`g`~~`q)`
`u()`	`T`	Sets the state to $u_{i+1} = \mathsf{TA}(u_i)$ and returns $\mathsf{GA}(u_i)$.	amortized constant
`u.discard(z)`[2]	`void`	post: The state of `u` is identical to that produced by `z` consecutive calls to `u()`.	no worse than the complexity of `z` consecutive calls to `u()`
`x == y`	`bool`	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 $S_x = S_y$; returns `false` otherwise.	$\mathscr{O}$(size of state)
`x != y`	`bool`	`!(x == y)`	$\mathscr{O}$(size of state)

[1]This constructor (as well as the corresponding `seed()` function below) may be particularly useful to applications requiring a large number of independent random sequences.

[2]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 to `u()`.

expression	return type	pre/post-condition	complexity
`os << x`	reference to the type of `os`	With `os`.*fmtflags* set to `ios_base::dec|ios_base::fixed|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.	$\mathscr{O}$(size of state)
`is >> v`	reference to the type of `is`	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` [lib.iostate.flags]). pre: The textual representation 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.	$\mathscr{O}$(size of state)

4 `X` shall satisfy the requirements of uniform random number generator [26.4.1.2] as well as of `CopyConstructible` [lib.copyconstructible] and of `Assignable` [lib.container.requirements]. Copy construction and assignment shall each be of complexity $\mathscr{O}$(size of state).

5 ~~If `Gen` is an arithmetic type [basic.fundamental], constructors instantiated from `template <class Gen> X(Gen& g)` as well as member functions instantiated from `template <class Gen> void seed(Gen& g)` shall have the same effect as `X(static_cast<Gen>(g))`.~~ [*Note:* ~~The cast makes `g` an rvalue, unsuitable for binding to a reference, to ensure that overload resolution will select the version of `seed` that takes a single integer argument instead of the version that takes a reference to a function object.~~ *— end note*]

6 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`.

26.4.1.4 Random number engine adaptor requirements [lib.rand.req.adapt]

1 A *random number engine adaptor* 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 adapted in this way are termed *base engines* in this context. The terms *unary*, *binary*, and so on, may be used to characterize an adaptor depending on the number n of base engines that adaptor utilizes.

2 A class `X` satisfies the requirements of a random number engine adaptor if the expressions shown in table 3 are valid and have the indicated semantics, and if `X` and its associated types also satisfies all other requirements of this section 26.4.1.4. In that table and throughout this section,

a) $\mathtt{B}_i$ is the type of the i^{th} of `X`'s base engines, $1 \le i \le n$; and

b) $\mathtt{b}_i$ is a value of $\mathtt{B}_i$.

If `X` is unary, i is omitted and understood to be 1.

Table 3: Random number engine adaptor requirements

expression	return type	pre/post-condition	complexity
`X::base`*i*`_type`	$\mathtt{B}_i$	—	compile time
`X::base`*i*`()`	`const` $\mathtt{B}_i$`&`	Returns a reference to $\mathtt{b}_i$.	constant

3 `X` shall satisfy the requirements of random number engine [26.4.1.3], subject to the following:

a) The base engines of `X` are arranged in an arbitrary but fixed order, and~~, unless otherwise specified,~~ that order is consistently used whenever functions are applied to those base engines in turn.

b) The complexity of each function is at most the sum of the complexities of the corresponding functions applied to each base engine.

c) The state of `X` includes the state of each of its base engines. The size of `X`'s state is no less than the sum of the base engine sizes. Copying `X`'s state (*e.g.*, during copy construction or copy assignment), includes copying, in turn, each base engine of `X`.

d) The textual representation of `X` includes, in turn, the textual representation of each of its base engines.

e) When `X::X` is invoked with no arguments, each of `X`'s base engines is constructed, in turn, as if by its respective default constructor. When `X::X` is invoked with an ~~`unsigned long`~~`X::result_type` value s, each of `X`'s base engines is constructed, in turn, with the next available value from the list $s+0, s+1, \ldots$. When `X::X` is invoked with ~~a zero-argument function object~~an argument of type `seed_seq`, each of `X`'s base engines is constructed, in turn, with that ~~function~~ object as argument. [*Note:* ~~This permits the function object to accumulate side effects.~~ — *end note*]

4 `X` shall have one additional constructor with n or more parameters such that the type of parameter i, $1 \le i \le n$, is `const` $\mathtt{B}_i$`&` and such that all remaining parameters, if any, have default values. The constructor shall construct `X`, initializing each of its base engines, in turn, with a copy of the value of the corresponding argument.

26.4.1.5 Random number distribution requirements [lib.rand.req.dist]

1 A class `X` satisfies the requirements of a *random number distribution* if the expressions shown in table 4 are valid and have the indicated semantics, and if `X` and its associated types also satisfies all other requirements of this section 26.4.1.5. In that table and throughout this section,

a) `T` is the type named by `X`'s associated `result_type`;

b) `P` is the type named by `X`'s associated `param_type`;

c) `u` is a value of `X` and `x` is a (possibly const) value of `X`;

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 `u`'s `operator()`, as determined by the current values of `u`'s parameters;

e) `p` is a value of `P`;

f) `e` is an lvalue of an arbitrary type that satisfies the requirements of a uniform random number generator [26.4.1.2];

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 [lib.strings] and [lib.input.output].

2 The specification of each random number distribution identifies an associated mathematical *probability density function* $p(z)$ or an associated *discrete probability function* $P(z_i)$. Such functions are typically expressed using certain externally-supplied quantities known as the *parameters of the distribution*. Such distribution parameters are identified in this context by writing, for example, $p(z\,|\,a,b)$ or $P(z_i\,|\,a,b)$, to name specific parameters, or by writing, for example, $p(z\,|\,\{\mathtt{p}\})$ or $P(z_i\,|\,\{\mathtt{p}\})$, to denote a distribution's parameters `p` taken as a whole.

Table 4: Random number distribution requirements

expression	return type	pre/post-condition	complexity
`X::result_type`	`T`	`T` is an arithmetic type.	compile-time
`X::param_type`	`P`	—	compile-time
`X(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
`u.reset()`	`void`	Subsequent uses of `u` do not depend on values produced by `e` prior to invoking `reset`.	constant
`x.param()`	`P`	Returns a value `p` such that `X(p).param() == p`.	no worse than the complexity of `X(p)`
`u.param(p)`	`void`	post: `u.param() == p`.	no worse than the complexity of `X(p)`
`u(e)`	`T`	With `p` = `u.param()`, the sequence of numbers returned by successive invocations with the same object `e` is randomly distributed according to the associated $p(z\,|\,\{\mathtt{p}\})$ or $P(z_i\,|\,\{\mathtt{p}\})$ function.	amortized constant number of invocations of `e`

expression	return type	pre/post-condition	complexity
`u(e,p)`	`T`	The sequence of numbers returned by successive invocations with the same objects `e` and `p` is randomly distributed according to the associated $p(z \mid \{\texttt{p}\})$ or $P(z_i \mid \{\texttt{p}\})$ function.	—
`x.min()`	`T`	Returns `glb`.	constant
`x.max()`	`T`	Returns `lub`.	constant
`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.	—
`is >> u`	reference to the type of `is`	Restores from `is` the parameters and additional internal data of `u`. If bad input is encountered, ensures that `u` is unchanged by the operation and calls `is.setstate(ios::failbit)` (which may throw `ios::failure` [lib.iostate.flags]). 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.	—

3 `X` shall satisfy the requirements of `CopyConstructible` [lib.copyconstructible] and `Assignable` [lib.container.requirements].

4 The sequence of numbers produced by repeated invocations of `x(e)` shall be independent of any invocation of `os << x` or of any `const` member function of `X` between any of the invocations `x(e)`.

5 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(e)` shall produce the same sequence of numbers as would repeated invocations of `x(e)`.

6 It is unspecified whether `X::param_type` is declared as a (nested) `class` or via a `typedef`. In this subclause 26.4, declarations of `X::param_type` are in the form of `typedef`s only for convenience of exposition.

7 `P` shall satisfy the requirements of `CopyConstructible`, `Assignable`, and `EqualityComparable` [lib.equalitycomparable].

8 For each of the constructors of `X` 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 `X` that return values corresponding to parameters of the distribution, `P` shall have a corresponding member function with the identical name, type, and semantics.

26.4.2 Header `<random>` synopsis [lib.rand.synopsis]

```
namespace std {
  // [26.4.3.1] Class template linear_congruential_engine
  template <class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine;

  // [26.4.3.2] Class template mersenne_twister_engine
  template <class UIntType, ~~int~~size_t w, ~~int~~size_t n, ~~int~~size_t m, ~~int~~size_t r,
            UIntType a, ~~int~~size_t u, ~~int~~size_t s,
            UIntType b, ~~int~~size_t t,
            UIntType c, ~~int~~size_t l>
    class mersenne_twister_engine;

  // [26.4.3.3] Class template subtract_with_carry_engine
  template <class UIntType, ~~IntType m~~size_t w, ~~int~~size_t s, ~~int~~size_t r>
    class subtract_with_carry_engine;

  ~~// [lib.rand.eng.sub1] Class template subtract_with_carry_01_engine~~
  ~~template <class RealType, int w, int s, int r>~~
    ~~class subtract_with_carry_01_engine;~~

  // [26.4.4.1] Class template discard_block_engine
  template <class Engine, ~~int~~size_t p, ~~int~~size_t r>
    class discard_block_engine;

  // [26.4.4.2] Class template random_bits_engine
  template <class Engine, size_t w, class UIntType>
    class random_bits_engine;

  // [26.4.4.3] Class template shuffle_order_engine
  template <class Engine, ~~int~~size_t k>
    class shuffle_order_engine;

  // [26.4.4.4] Class template xor_combine_engine
  template <class Engine1, ~~int~~size_t s1, class Engine2, ~~int~~size_t s2=0u>
    class xor_combine_engine;

  // [26.4.5] Engines and engine adaptors with predefined parameters
  typedef see below minstd_rand0;
  typedef see below minstd_rand;
  typedef see below mt19937;
  typedef see below ~~ranlux_base_01~~ranlux24_base;
  typedef see below ~~ranlux64_base_01~~ranlux48_base;
  ~~typedef see below ranlux3;~~
```

```
~~typedef see below ranlux4;~~
typedef see below ~~ranlux3_01~~ranlux24;
typedef see below ~~ranlux4_01~~ranlux48;
typedef see below knuth_b;

// [26.4.6] Class random_device
class random_device;

// [26.4.7.1] Class seed_seq
class seed_seq;

// [26.4.7.2] Function template generate_canonical
template<class result_type, class UniformRandomNumberGenerator>
  result_type generate_canonical(UniformRandomNumberGenerator& g);

// [26.4.8.1.1] Class template uniform_int_distribution
template <class IntType = int>
  class uniform_int_distribution;

// [26.4.8.1.2] Class template uniform_real_distribution
template <class RealType = double>
  class uniform_real_distribution;

// [26.4.8.2.1] Class bernoulli_distribution
class bernoulli_distribution;

// [26.4.8.2.2] Class template binomial_distribution
template <class IntType = int>
  class binomial_distribution;

// [26.4.8.2.3] Class template geometric_distribution
template <class IntType = int>
  class geometric_distribution;

// [26.4.8.2.4] Class template negative_binomial_distribution
template <class IntType = int>
  class negative_binomial_distribution;

// [26.4.8.3.1] Class template poisson_distribution
template <class IntType = int>
  class poisson_distribution;

// [26.4.8.3.2] Class template exponential_distribution
template <class RealType = double>
  class exponential_distribution;

// [26.4.8.3.3] Class template gamma_distribution
template <class RealType = double>
  class gamma_distribution;
  // [26.4.8.3.4] Class template weibull_distribution
  template <class RealType = double>
    class weibull_distribution;

  // [26.4.8.3.5] Class template extreme_value_distribution
  template <class RealType = double>
    class extreme_value_distribution;

  // [26.4.8.4.1] Class template normal_distribution
  template <class RealType = double>
    class normal_distribution;

  // [26.4.8.4.2] Class template lognormal_distribution
  template <class RealType = double>
    class lognormal_distbution;

  // [26.4.8.4.3] Class template chi_squared_distribution
  template <class RealType = double>
    class chi_squared_distribution;

  // [26.4.8.4.4] Class template cauchy_distribution
  template <class RealType = double>
    class cauchy_distribution;

  // [26.4.8.4.5] Class template fisher_f_distribution
  template <class RealType = double>
    class fisher_f_distribution;

  // [26.4.8.4.6] Class template student_t_distribution
  template <class RealType = double>
    class student_t_distribution;

  // [26.4.8.5.1] Class template discrete_distribution
  template <class IntType = int>
    class discrete_distribution;

  // [26.4.8.5.2] Class template piecewise_constant_distribution
  template <class RealType = double>
    class piecewise_constant_distribution;

  // [26.4.8.5.3] Class template general_pdf_distribution
  template <class RealType = double>
    class general_pdf_distribution;
} // namespace std
```

26.4.3 Random number engine class templates [lib.rand.eng]

1 Except where specified otherwise, the complexity of all functions specified in the following sections is constant.

2 Except as required by table 2, no function described in this section 26.4.3 throws an exception.

3 The class templates specified in this section 26.4.3 satisfy the requirements of random number engine [26.4.1.3]. Descriptions are provided here only for operations on the engines that are not described in those requirements or for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment operators, for the `bits_of_randomness` member, and for equality and inequality operators are not shown in the synops~~i~~es.

26.4.3.1 Class template `linear_congruential_engine` [lib.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 <class UIntType, UIntType a, UIntType c, UIntType m>
class linear_congruential_engine
{
public:
  // types
  typedef UIntType result_type;

  // ~~parameter values and~~ engine characteristics
  static const ~~UIntType~~result_type multiplier = a;
  static const ~~UIntType~~result_type increment = c;
  static const ~~UIntType~~result_type modulus = m;
  static const result_type min = c == 0u ? 1u: 0u;
  static const result_type max = m - 1u ;
  static const ~~unsigned long~~result_type default_seed = 1u;

  // constructors and seeding functions
  explicit linear_congruential_engine(~~unsigned long~~result_type s = default_seed);
  ~~template <class Gen>~~ explicit linear_congruential_engine(~~Gen& g~~seed_seq& q);
  void seed(~~unsigned long~~result_type s = default_seed);
  ~~template <class Gen>~~ void seed(~~Gen& g~~seed_seq& q);

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

2 The template parameter `UIntType` shall denote an unsigned integral type large enough to store values as large as $m-1$. If the template parameterm is 0, the modulus m used throughout this section 26.4.3.1 is `numeric_limits<`~~`UIntType`~~`result_type>::max()` plus 1. [*Note:* The result need not be representable as a value of type ~~`UIntType`~~`result_type`. —*end note*] Otherwise, the following relations shall hold: `a` < `m` and `c` < `m`.

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

```
explicit linear_congruential_engine(~~unsigned long~~result_type s = default_seed);
```

4 *Effects:* Constructs a `linear_congruential_engine` object. If $c \bmod m$ is 0 and $\mathtt{s} \bmod m$ is 0, sets the engine's state to 1, otherwise sets the engine's state to $\mathtt{s} \bmod m$.

```
~~template <class Gen>~~ explicit linear_congruential_engine(~~Gen& g~~seed_seq& q);
```

5 *Effects:* Constructs a `linear_congruential_engine` object. With ~~γ = `g()` mod m,~~$k = \left\lceil \frac{\log_2 m}{32} \right\rceil$ and a an array (or equivalent) of length $k+3$, invokes `q.randomize(`$a+0$`, `$a+k+3$`)` and then computes $S = \left(\sum_{j=0}^{k-1} a_{j+3} \cdot 2^{32j}\right)$ mod m. ~~i~~If c mod m is 0 and ~~γ~~S is 0, sets the engine's state to 1, else sets the engine's state to ~~γ~~S.

6 *Complexity:* ~~Exactly one invocation of `g`.~~

26.4.3.2 Class template `mersenne_twister_engine` [lib.rand.eng.mers]

1 A `mersenne_twister_engine` random number engine[3] 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, 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} xor $(Y \,\mathsf{rshift}\, 1)$ xor α.

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 $z_1 = X_i \,\mathsf{xor}\, (X_i \,\mathsf{rshift}\, u)$.

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 <class UIntType, ~~int~~size_t w, ~~int~~size_t n, ~~int~~size_t m, ~~int~~size_t r,
            UIntType a, ~~int~~size_t u, ~~int~~size_t s,
            UIntType b, ~~int~~size_t t,
            UIntType c, ~~int~~size_t l>
class mersenne_twister_engine
{
public:
  // types
  typedef UIntType} result_type;

  // ~~parameter values and~~ engine characteristics
  static const ~~int~~size_t word_size = w;
  static const ~~int~~size_t state_size = n;
  static const ~~int~~size_t shift_size = m;
  static const ~~int~~size_t mask_bits = r;
  static const UIntType xor_mask = a;
  static const ~~int~~size_t tempering_u = u;
```

[3] 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.

```
    static const ~~int~~size_t tempering_s = s;
    static const UIntType tempering_b = b;
    static const ~~int~~size_t tempering_t = t;
    static const UIntType tempering_c = c;
    static const ~~int~~size_t tempering_l = l;
    static const result_type min = 0;
    static const result_type max = 2^w − 1;
    static const ~~unsigned long~~result_type default_seed = 5489u;

    // constructors and seeding functions
    explicit mersenne_twister_engine(~~unsigned long~~result_type value = default_seed);
    ~~template <class Gen>~~ explicit mersenne_twister_engine(~~Gen& g~~seed_seq& q);
    void seed(~~unsigned long~~result_type value = default_seed);
    ~~template <class Gen>~~ void seed(~~Gen& g~~seed_seq& q);

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

4 The ~~template parameter `UIntType` shall denote an unsigned integral type large enough to store values up to $2^{\mathtt{w}} - 1$. Also, the~~ following relations shall hold: $1 \le \mathtt{m} \le \mathtt{n}$; $0 \le \mathtt{r}, \mathtt{u}, \mathtt{s}, \mathtt{t}, \mathtt{l} \le \mathtt{w}$$\le$ `numeric_limits<result_type>::digits`; $0 \le \mathtt{a}, \mathtt{b}, \mathtt{c} \le 2^{\mathtt{w}} - 1$.

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(~~unsigned long~~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-n}~~X_i to

$$\left[1812433253 \cdot \left(X_{i-1} \operatorname{xor} \left(X_{i-1} \operatorname{rshift} (w-2)\right)\right) + i \bmod n\right] \bmod 2^w .$$

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

`~~template <class Gen>~~ explicit mersenne_twister_engine(~~Gen& g~~seed_seq& q);`

8 *Effects:* Constructs a `mersenne_twister_engine` object. ~~Given the values ... obtained by successive invocations of g, sets ... to ..., respectively.~~With $k = \lceil w/32 \rceil$ and a an array (or equivalent) of length $n \cdot k$, invokes `q.randomize(`$a+0$`, `$a+n\cdot k$`)` 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 each of the resulting X_i is 0, changes X_{-n} to 2^{w-1}.

9 *Complexity:* ~~Exactly n invocations of g.~~

26.4.3.3 Class template `subtract_with_carry_engine` [lib.rand.eng.sub]

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

2 The state 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 x_i additionally consists of an integer c (known as the *carry*) whose value is either 0 or 1.

3 ~~The transition algorithm is a modular linear function of the form . . . .~~ 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 $\mathsf{TA}(x_i) = (a \cdot x_i) \bmod b$, where b is of the form $m^r - m^s + 1$ and $a = b - \frac{b-1}{m}$. — *end note*]

4 The generation algorithm ~~yields the last~~is given by $\mathsf{GA}(x_i) = y$, where y is the value ~~of Y mod m~~ produced as a result of advancing the engine's state as described above.

```
template <class UIntType, ~~IntType m~~size_t w, ~~int~~size_t s, ~~int~~size_t r>
class subtract_with_carry_engine
{
public:
  // types
  typedef UIntType result_type;

  // ~~parameter values and~~ engine characteristics
  static const ~~IntType modulus = m~~size_t word_size = w;
  static const ~~int~~size_t short_lag = s;
  static const ~~int~~size_t long_lag = r;
  static const result_type min = 0;
  static const result_type max = ~~m - 1~~m − 1;
  static const ~~unsigned long~~result_type default_seed = 19780503u;

  // constructors and seeding functions
  explicit subtract_with_carry_engine(~~unsigned long~~result_type value = default_seed);
  ~~template <class Gen>~~ explicit subtract_with_carry_engine(~~Gen& g~~seed_seq& q);
  void seed(~~unsigned long~~result_type value = default_seed);
  ~~template <class Gen>~~ void seed(~~Gen& g~~seed_seq& q);

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

5 ~~The template parameter `IntType` shall denote a signed integral type large enough to store values up to `m`.~~

6 The following relations shall hold: $0 < \mathtt{s} < \mathtt{r}$, and $0 < \mathtt{w} \leq$ `numeric_limits<result_type>::digits`.

7 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(~~unsigned long~~result_type value = default_seed);
```

8 *Effects:* Constructs a `subtract_with_carry_engine` object ~~as if the constructor `subtract_with_carry_engine(g)` had been invoked, where `g` had been freshly constructed:~~. Sets the values of $X_{-r}, \ldots, X_{-1}$, in that order, as required below. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.

9 *Required behavior:* First construct `e`, a `linear_congruential_engine` object, as if by the following definition:

```
linear_congruential_engine<~~unsigned long,~~result_type
                           40014u,0u,2147483563u> ~~g~~e(value == 0u ? default_seed : value);
```

To set an X_k, use new values $z_0, \ldots, z_{n-1}$ obtained from n successive invocations of `e` taken modulo 2^{32}. Set X_k to $\left(\sum_{j=0}^{n-1} z_j \cdot 2^{32j}\right) \bmod m$. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.

10 *Complexity:* Exactly $n \cdot$ `r` invocations of ~~g~~e.

`~~template <class Gen>~~ explicit subtract_with_carry_engine(~~Gen& g~~seed_seq& q);`

11 *Effects:* Constructs a `subtract_with_carry_engine` object. With ~~..., sets ...~~$k = \lceil w/32 \rceil$ and a an array (or equivalent) of length $r \cdot k$, invokes `q.randomize(`$a+0$`, `$a+r \cdot k$`)` 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.

12 *Complexity:* ~~Exactly `r*n` invocations of `g`.~~

26.4.3.4 Class template `subtract_with_carry_01_engine` [TO BE DELETED] [lib.rand.eng.sub1]

Delete the entirety of this subsection.

1 A `subtract_with_carry_01_engine` random number engine produces floating-point random numbers. The state x_i of a `subtract_with_carry_01_engine` object `x` is of size $\mathcal{O}$(r), and consists of a sequence X of r integer values $0 \le X_i < 2^w$; all subscripts applied to X are to be taken modulo r. The state x_i additionally consists of an integer c (known as the *carry*) whose value is either 0 or 1.

2 The transition algorithm is a modular linear function of the form $\mathsf{TA}(\mathsf{x}_i) = (a \cdot \mathsf{x}_i) \bmod p$, where p is of the form $2^{wr} - 2^{ws} + 1$ and $a = p - \frac{p-1}{b}$. The state transition is performed as follows:

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

b) Set X_i to $Y \bmod 2^w$. Set c to 1 if $Y < 0$, otherwise set c to 0.

[*Note:* This state transition algorithm is identical to that used by a `subtract_with_carry_engine` [26.4.3.3] with $m = 2^w$. — *end note*]

3 The generation algorithm is $\mathsf{GA}(\mathsf{x}_i) = T \cdot 2^{-w} + \varepsilon$ where T is the last value of $Y \bmod 2^w$ produced as a result of advancing the engine's state as described above and ε is $2^{-(w+2)}$. [*Note:* This guarantees that the values produced will lie in the required open interval $(0,1)$. — *end note*]

```
template <class RealType, int w, int s, int r>
class subtract_with_carry_01_engine
{
public:
  // types
  typedef RealType result_type;

  // ~~parameter values and~~ engine characteristics
  static const int word_size = w;
```

```
    static const int short_lag = s;
    static const int long_lag = r;
    static const int min = 0;
    static const int max = 1;
    static const unsigned long default_seed = 19780503u;

    // constructors and seeding functions
    explicit subtract_with_carry_01_engine(unsigned long value = default_seed);
    template <class Gen> explicit subtract_with_carry_01_engine(Gen& g);
    void seed(unsigned long value = default_seed);
    template <class Gen> void seed(Gen& g);

    // generating functions
    result_type operator()();
  };
```

4 The following relations shall hold: $0 < \texttt{s} < \texttt{r}$ and $\texttt{w} < -\texttt{numeric\_limits<RealType>::min\_exponent} - 2$. [*Note:* The latter relation ensures that ε, used above, is representable as a non-zero value of `result_type`. —*end note*]

5 The textual representation consists of the textual representations of $X_{i-r}, \ldots, X_{i-1}$, in that order, followed by c. The textual representation of each X_k consists of the sequence of $n = \lfloor(\texttt{w}+31)/32\rfloor$ integer numbers z_j, in the order $z_0, \ldots, z_{n-1}$, defined such that $\sum_{j=0}^{n-1} z_j \cdot 2^{32j} = X_k$. [*Note:* This algorithm ensures that only integer numbers representable in 32 bits are written. —*end note*]

```
explicit subtract_with_carry_01_engine(unsigned long value = default_seed);
```

6 *Effects:* Constructs a `subtract_with_carry_01_engine` object as if the constructor `subtract_with_carry_-01_engine(g)` had been invoked, where g had been freshly constructed as if by the following definition:

```
linear_congruential_engine<unsigned long,40014,0,2147483563> g(value == 0u ? default_seed
                                                                            : value);
```

```
template <class Gen> explicit subtract_with_carry_01_engine(Gen& g);
```

7 *Effects:* Constructs a `subtract_with_carry_01_engine` object. With n as above, sets the values of $X_{-r}, \ldots, X_{-1}$, in that order, as follows. To set X_k, obtain values $z_0, \ldots, z_{n-1}$ by successive invocations of g taken modulo 2^{32}, and set X_k to $\sum_{j=0}^{n-1} z_j \cdot 2^{32j}$. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.

8 *Complexity:* Exactly $n \cdot \texttt{r}$ invocations of g.

26.4.4 Random number engine adaptor class templates [lib.rand.adapt]

1 Except where specified otherwise, the complexity of all functions specified in the following sections is constant.

2 Except as required by table 2, no function described in this section 26.4.4 throws an exception.

3 The class templates specified in this section 26.4.4 satisfy the requirements of random number engine adaptor [26.4.1.4]. Descriptions are provided here only for operations on the engine adaptors that are not described in those requirements or

for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment operators, and for equality and inequality operators are not shown in the synopses.

26.4.4.1 Class template `discard_block_engine` [lib.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 <class Engine, ~~int~~size_t p, ~~int~~size_t r>
class discard_block_engine
{
public:
  // types
  typedef Engine base_type;
  typedef typename base_type::result_type result_type;

  // ~~parameter values and~~ engine characteristics
  static const ~~int~~size_t block_size = p;
  static const ~~int~~size_t used_block = r;
  static const result_type min = base_type::min;
  static const result_type max = base_type::max;

  // constructors and seeding functions
  discard_block_engine();
  explicit discard_block_engine(const base_type& urng);
  explicit discard_block_engine(~~unsigned long~~result_type s);
  ~~template <class Gen>~~ explicit discard_block_engine(~~Gen& g~~seed_seq& q);
  void seed();
  void seed(~~unsigned long~~result_type s);
  ~~template <class Gen>~~ void seed(~~Gen& g~~seed_seq& q);

  // generating functions
  result_type operator()();
  void discard(size_t z);

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

private:
  base_type e;    // exposition only
  int n;          // exposition only
};
```

4 The following relations shall hold: $1 \le$ `r` $\le$ `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 26.4.1.4, each constructor that is not a copy constructor sets `n` to 0.

~~`const base_type& base() const;`~~

7 *Returns:* ~~A reference to `e`.~~

26.4.4.2 Class template `random_bits_engine` [lib.rand.adapt.rbits]

1 A `random_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 a `random_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 With $R =$ `e.max - e.min + 1` and $m = \lfloor \log_2 R \rfloor$, the transition algorithm is carried out by invoking `e()` as often as needed to obtain $k = \lceil w/m \rceil$ values less than $2^m +$ `e.min`.

3 The generation algorithm uses the values produced while advancing the state as described above to yield a `result_type` quantity S obtained as if by the following algorithm:

```
S = 0;
for (n = 0; n < w; n += m)  {
  do u = e() - e.min; while (u ≥ 2^m);
  S = 2^m · S + u;
}
S = S mod 2^w;

template <class Engine, size_t w, class UIntType>
class random_bits_engine
{
public:
  // types
  typedef Engine base_type;
  typedef UIntType result_type;

  // engine characteristics
  static const result_type min = 0;
  static const result_type max = 2^w − 1;

  // constructors and seeding functions
  random_bits_engine();
  explicit random_bits_engine(const base_type& urng);
  explicit random_bits_engine(result_type s);
  explicit random_bits_engine(seed_seq& q);
  void seed();
  void seed(result_type s);
  void seed(seed_seq& q);

  // generating functions
```

```
    result_type operator()();
    void discard(size_t z);

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

  private:
    base_type e;     // exposition only
  };
```

4 The following relations shall hold: $0 < w \leq$ `numeric_limits<result_type>::digits`. Additionally, `numeric_limits<base_type::result_type>::digits` $\leq$ `numeric_limits<result_type>::digits` shall hold.

5 The textual representation consists of the textual representation of `e`.

26.4.4.3 Class template `shuffle_order_engine` [lib.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 as $\left\lfloor \frac{k\cdot(Y-b_{\min})}{b_{\max}-b_{\min}+1} \right\rfloor$, if e is integer-valued, or as $\lfloor k\cdot Y \rfloor$, if e is real-valued.

b) Set Y to V_j and then set V_j to `b()`.

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

```
  template <class Engine, ~~int~~size_t k>
  class shuffle_order_engine
  {
  public:
    // types
    typedef Engine base_type;
    typedef typename base_type::result_type result_type;

    // ~~parameter values and~~ engine characteristics
    static const ~~int~~size_t table_size = k;
    static const result_type min = base_type::min;
    static const result_type max = base_type::max;

    // constructors and seeding functions
    shuffle_order_engine();
    explicit shuffle_order_engine(const base_type& urng);
    explicit shuffle_order_engine(~~unsigned long~~result_type s);
    ~~template <class Gen>~~ explicit shuffle_order_engine(~~Gen& g~~seed_seq& q);
    void seed();
    void seed(~~unsigned long~~result_type s);
    ~~template <class Gen>~~ void seed(~~Gen& g~~seed_seq& q);
    // generating functions
    result_type operator()();
    void discard(size_t z);

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

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

4 The following relation shall hold: $1 \leq$ `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 26.4.1.4, 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()`.

~~`const base_type& base() const;`~~

7 *Returns:* ~~A reference to e.~~

26.4.4.4 Class template `xor_combine_engine` [lib.rand.adapt.xor]

1 An `xor_combine_engine` random number engine adaptor produces random numbers from two integer-valued base engines `e1` and `e2` by merging their left-shifted random values via bitwise exclusive-or. The state x_i of an `xor_combine_engine` engine adaptor object `x` consists of the states $e1_i$ and $e2_i$ of its base engines. The size of the state is the size of the state of `e1` plus the size of the state of `e2`.

2 The transition algorithm advances, in turn, the state of each base engine.

3 The generation algorithm is $\mathsf{GA}(x_i) = ($~~`e1()`~~v_1 lshift_w `s1`$)$ xor $($~~`e2()`~~v_2 lshift_w `s2`$)$, where w denotes the value of `numeric_limits<result_type>::digits` and v_1 and v_2, respectively, denote the values of `(e1()-e1.min)` and `(e2()-e2.min)`.

```
  template <class Engine1, ~~int~~size_t s1, class Engine2, ~~int~~size_t s2=0u>
  class xor_combine_engine
  {
  public:
    // types
    typedef Engine1 base1_type;
    typedef Engine2 base2_type;
    typedef see below result_type;

    // ~~parameter values and~~ engine characteristics
    static const ~~int~~size_t shift1 = s1;
    static const ~~int~~size_t shift2 = s2;
    static const result_type min = 0;
    static const result_type max = see below;

    // constructors and seed functions
    xor_combine_engine();
    xor_combine_engine(const base1_type & urng1, const base2_type & urng2);
    xor_combine_engine(~~unsigned long~~result_type s);
    ~~template <class Gen>~~ explicit xor_combine_engine(~~Gen& g~~seed_seq& q);
    void seed();
    void seed(result_type s);
    ~~template <class Gen>~~ void seed(~~Gen& g~~seed_seq& q);

    // generating functions
    result_type operator()();
    void discard(size_t z);

    // property functions
    const base1_type& base1() const;
    const base2_type& base2() const;

  private:
    base1_type e1;    // exposition only
    base2_type e2;    // exposition only
  };
```

4 The following relations shall hold: $\mathtt{s1} \geq \mathtt{s2} \geq 0$.

5 [*Note:* An `xor_combine_engine` engine adaptor that fails to observe the following recommendations may have significantly worse uniformity properties than either of the base engines it is based on:

a) While two shift values (template parameters `s1` and `s2`) are provided for simplicity of interface, it is advisable that ~~at most one of these values~~`s2` be ~~non-~~zero. (If ~~both `s1` and~~ `s2` ~~are~~is non-zero then the low bits will always be 0.)

b) It is also advisable for ~~the unshifted base engine~~e2's `max` to be $2^n - 1 - \mathtt{min}$ for some non-negative integer n, and for the shift ~~applied to the other base engine~~value `s1` to be no greater than that n.

— *end note*]

6 Both `Engine1::result_type` and `Engine2::result_type` shall denote (possibly different) unsigned integral types. The member `result_type` shall denote either the type `Engine1::result_type` or the type `Engine2::result_type`, whichever provides the most storage according to clause [basic.fundamental].

7 With ~~w as above, and given the unsigned integer values~~

a) $m_1 = min($`Engine1::max` $-$ `Engine1::min` ~~$)$ lshift$_w$ $($s1 $-$ s2$)$,~~ $, 2^{w-\mathtt{s1}} - 1)$,

b) $m_2 = min($`Engine2::max` $-$ `Engine2::min`$, 2^{w-\mathtt{s2}} - 1)$,

c) ~~$A = m_1$ bitand m_2,~~

d) ~~$B = m_1$ bitor m_2,~~ and

e) ~~$C = \ldots$~~ $s = \mathtt{s1} - \mathtt{s2}$, the value of the member `max` is ~~$(C$ bitor $B)$ lshift$_w$ (s1 − s2).~~$M(m_1,m_2,s)$ lshift$_w$ s2, where $M(a,b,d)$ is defined as follows:

If $a = 0$ or $b < 2^d$, define $M(a,b,d)$ as $a \cdot 2^d + b$.

Otherwise, let t and u denote the greater and the lesser, respectively, of $a \cdot 2^d$ and b. With $p = \lfloor \log_2 u \rfloor$, if $k = \lfloor t/2^p \rfloor$ is odd, define $M(a,b,d)$ as $(k+1) \cdot 2^p - 1$.

Otherwise, if $a \cdot 2^d \geq b$, define $M(a,b,d)$ as $(k+1) \cdot 2^p + M((t \bmod 2^p)/2^d, u \bmod 2^p, d)$.

Otherwise, define $M(a,b,d)$ as $(k+1) \cdot 2^p + M((u \bmod 2^p)/2^d, t \bmod 2^p, d)$.

8 The textual representation consists of the textual representation of `e1` followed by the textual representation of `e2`.

~~`const base_type& base1() const;`~~

9 *Returns:* ~~A reference to e1.~~

~~`const base_type& base2() const;`~~

10 *Returns:* ~~A reference to e2.~~

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

```
typedef linear_congruential_engine<~~unsigned long~~uint_fast32_t, 16807, 0, 2147483647>
        minstd_rand0;
```

1 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type `minstd_rand0` shall produce the value 1043618065.

```
typedef linear_congruential_engine<~~unsigned long~~uint_fast32_t, 48271, 0, 2147483647>
        minstd_rand;
```

2 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type `minstd_rand` shall produce the value 399268537.

```
typedef mersenne_twister_engine<~~unsigned long~~uint_fast32_t,
                                32,624,397,31,0x9908b0df,11,7,0x9d2c5680,15,0xefc60000,18>
        mt19937;
```

3 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type `mt19937` shall produce the value 4123659995.

```
typedef subtract_with_carry~~_01~~_engine<~~float~~uint_fast32_t, 24, 10, 24>
        ranlux~~_base_01~~24_base;
```

4 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type ranlux~~_base_01~~`24_base` shall produce the value 7937952.

```
typedef subtract_with_carry~~_01~~_engine<~~double~~uint_fast64_t, 48, 5, 12>
        ranlux~~64_base_01~~48_base;
```

5 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type ranlux~~64_base_01~~`48_base` shall produce the value ~~19211384363394 8~~61839128582725.

```
~~typedef discard_block_engine<subtract_with_carry_engine<unsigned long, (1<<24), 10, 24>,~~
                                    ~~223, 24>~~
          ~~ranlux3;~~
```

6 *Required behavior:* ~~The 10000th consecutive invocation of a default-constructed object of type `ranlux3` shall produce the value 5957620.~~

```
~~typedef discard_block_engine<subtract_with_carry_engine<unsigned long, (1<<24), 10, 24>,~~
                                    ~~389, 24>~~
          ~~ranlux4;~~
```

7 *Required behavior:* ~~The 10000th consecutive invocation of a default-constructed object of type `ranlux4` shall produce the value 8587295.~~

```
typedef discard_block_engine<ranlux~~_base_01~~24_base, 223, ~~24~~23>
          ranlux~~3_01~~24;
```

8 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type ranlux~~3_01~~24 shall produce the value ~~5957620~~9901578.

```
typedef discard_block_engine<ranlux~~_base_01~~48_base, 389, ~~24~~11>
          ranlux~~4_01~~48;
```

9 *Required behavior:* The 10000th consecutive invocation of a default-constructed object of type ranlux~~4_01~~48 shall produce the value ~~8587295~~249142670248501.

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

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

26.4.6 Class `random_device` [lib.rand.device]

1 A `random_device` uniform random number generator produces non-deterministic random numbers. It satisfies the requirements of uniform random number generator [26.4.1.2]. ~~Descriptions are provided here only for operations that are not described in those requirements or for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment operators, and for equality and inequality operators are not shown in the synopsis.~~

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

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

  // generator characteristics
  static const result_type min = see below;
  static const result_type max = see below;
```

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

    // generating functions
    result_type operator()();

    // property functions
    double entropy() const;

  private:
    random_device(const random_device& );
    void operator=(const random_device& );
  };
```

3 The values of the `min` and `max` members are identical to the values returned by `numeric_limits<result_type>::min()` and `numeric_limits<result_type>::max()`, respectively.

`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.[4)]

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[5)] for the random numbers returned by `operator()`, in the range `min`~~`()`~~ to $\log_2($`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.

26.4.7 Utilities [lib.rand.util]

26.4.7.1 Class `seed_seq` [lib.rand.util.seedseq]

1 An object of type `seed_seq` consumes a sequence of integer-valued data and produces a fixed number of unsigned integer values, $0 \le i < 2^{32}$, based on the consumed data. [*Note:* Such an object provides a mechanism to avoid replication of streams of random variates. This can be useful in applications requiring large numbers of random number engines. — *end note*]

4) The parameter is intended to allow an implementation to differentiate between different sources of randomness.
5) If a device has n states whose respective probabilities are $P_0, \ldots, P_{n-1}$, the device entropy S is defined as $S = -\sum_{i=0}^{n-1} P_i \cdot \log P_i$.

2 In addition to the requirements set forth below, instances of `seed_seq` shall meet the requirements of `CopyConstructible` [lib.copyconstructible] and of `Assignable` [lib.container.requirements].

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

  // constructors and reset functions
  seed_seq();
  template<class InputIterator> seed_seq(InputIterator begin, InputIterator end);

  // generating functions
  template<class RandomAccessIterator>
    void randomize(RandomAccessIterator begin, RandomAccessIterator end) const;

  // property functions
  size_t size() const;
  template<class OutputIterator> void get_seeds(OutputIterator dest) const;

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

`explicit seed_seq();`

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

4 *Throws:* Nothing.

`template<class InputIterator> seed_seq(InputIterator begin, InputIterator end);`

5 *Requires:* `InputIterator` shall satisfy the requirements of an input iterator [lib.input.iterator] such that `iterator_traits<InputIterator>::value_type` shall denote an integral type.

6 *Effects:* Constructs a `seed_seq` object by rearranging the bits of the supplied sequence `[begin,end)` into 32-bit units, as if by first concatenating all the n bits that make up the supplied sequence to initialize a single (possibly very large) unsigned binary number, b, and then carrying out the following algorithm:

```
for( v.clear(); $n$ > 0; $n$ -= 32 )
  v.push_back($b \bmod 2^{32}$), $b$ /= $2^{32}$;
```

```
template<class RandomAccessIterator>
void randomize(RandomAccessIterator begin, RandomAccessIterator end) const;
```

7 *Requires:* `RandomAccessIterator` shall meet the requirements of a random access iterator [lib.random.access.iterators] such that `iterator_traits<RandomAccessIterator>::value_type` shall denote an unsigned integral type capable of accommodating 32-bit quantities.

8 *Effects:* With $s =$ `v.size()` and $n =$ `end` $-$ `begin`, fills the supplied range [`begin`, `end`) according to the following

algorithm[6)] in which each operation is to be carried out modulo 2^{32}, each indexing operator applied to `begin` is to be taken modulo n, each indexing operator applied to `v` is to be taken modulo s, and $T(x)$ is defined as $x \operatorname{xor} (x \operatorname{rshift} 30)$:

a) Set `begin[0]` to $5489 + s$. Then, iteratively for $k = 1, \ldots, n-1$, set `begin[k]` to

$$1812433253 \cdot T(\texttt{begin[k-1]}) + k \,.$$

b) With m as the larger of s and n, transform each element of the range (possibly more than once): iteratively for $k = 0, \ldots, m-1$, set `begin[k]` to

$$(\texttt{begin[k]} \operatorname{xor} (1664525 \cdot T(\texttt{begin[k-1]}))) + \texttt{v[k]} + (k \bmod s) \,.$$

c) Transform each element of the range one last time, beginning where the previous step ended: iteratively for $k = m \bmod n, \ldots, n-1, 0, \ldots, (m-1) \bmod n$, set `begin[k]` to

$$(\texttt{begin[k]} \operatorname{xor} (1566083941 \cdot T(\texttt{begin[k-1]}))) - k \,.$$

9 *Throws:* Nothing.

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

10 *Returns:* The number of 32-bit units the object can deliver, as if by returning the result of `v.size()`.

```
template<class OutputIterator> void get_seeds(OutputIterator dest) const;
```

11 *Requires:* `OutputIterator` shall satisfy the requirements of an output iterator [lib.output.iterator] such that `iterator_traits<OutputIterator>::value_type` shall be assignable from `result_type`.

12 *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);
```

26.4.7.2 Function template `generate_canonical` [lib.rand.util.canonical]

1 Each function instantiated from the template described in this section 26.4.7.2 maps the result of ~~a single~~one or more invocations of a supplied uniform random number generator `g` to one member of ~~$\mathscr{I}$ (described below)~~the specified `result_type` such that, if the values g_i produced by ~~the generator~~`g` are uniformly distributed, the instantiation's results t_j are distributed as uniformly as possible ~~according to the uniformity requirements described~~as specified below.

2 [*Note:* Obtaining a value in ~~$\mathscr{I}$~~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*]

6) This algorithm is due to Dr. Makoto Matsumoto, with an improvement (approved by Dr. Matsumoto) suggested by Dr. Charles Karney.

3 ~~For purposes of this section let $\mathscr{I}$ consist of all values t_j of type `result_type` such that:~~ With ε as the value of `numeric_limits<result_type>::epsilon()`,

a) If `result_type` is a floating-point type [basic.fundamental], ~~`result_type(0)` $< t_j <$ `result_type(1)`~~ $\varepsilon \le t_j \le 1-\varepsilon$.

b) If `result_type` is a signed or unsigned integral type [basic.fundamental], `numeric_limits<result_type>::min()` $\le t_j \le$ `numeric_limits<result_type>::max()`.

```
template<class result_type, class UniformRandomNumberGenerator, size_t bits=1u>
result_type generate_canonical(UniformRandomNumberGenerator& g);
```

4 *Complexity:* Exactly ~~one~~$k = \lceil b/log_2 R \rceil$ invocations of `g`, where b[7] is the lesser of `numeric_limits<result_-type>::digits` and `bits` (but not less than one), and R is the value of `g.max - g.min + 1`.

5 *Required behavior:* ~~Let $|\mathscr{I}|$ denote the number of distinct values in $\mathscr{I}$ and let $|$g$|$ denote the number of distinct values that g is capable of producing. Finally, let x be the value resulting from the invocation of g.~~

a) ~~If $|$g$| = |\mathscr{I}|$, each distinct value produced by g shall correspond in an unspecified manner to a unique value from $\mathscr{I}$. The unique value corresponding to x shall be returned as the result of the function call.~~

b) ~~Otherwise, if $|$g$| < |\mathscr{I}|$, each distinct value that g can produce shall correspond in an unspecified manner to a unique subrange of values from $\mathscr{I}$. The subranges shall be contiguous and non-overlapping, and the number of values in each subrange shall differ by no more than one from the number of values in any other subrange. One value from the subrange corresponding to x shall be selected in an unspecified manner, and shall be returned as the result of the function call.~~

c) ~~Otherwise, $|$g$| > |\mathscr{I}|$ must hold, and the set of distinct values that g can produce shall be partitioned in an unspecified manner into $|\mathscr{I}|$ non-intersecting subsets, with the cardinalities of no two subsets differing by more than one. Each subset shall correspond in an unspecified manner to a unique value from $\mathscr{I}$. The unique value corresponding to the subset containing x shall be returned as the result of the function call.~~

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 `result_type`, if floating-point, otherwise using exact arithmetic.

6 *Returns:* ~~A value from $\mathscr{I}$, subject to the following required behavior~~With ε as above, and with M as the value of $1+$`numeric_limits<result_type>::max()`, either

$$\frac{S\cdot(1-2\varepsilon)}{R^k-1}+\varepsilon \text{ , if } \texttt{result\_type} \text{ is real-valued, or } \left\lfloor \frac{M\cdot(2S+1)}{2R^k} \right\rfloor \text{ , otherwise.}$$

7 *Throws:* What and when `g` throws.

[7] b is introduced to avoid any attempt to produce more bits of randomness than can be held in the `result_type`.

26.4.8 Random number distribution class templates [lib.rand.dist]

1 The classes and class templates specified in this section 26.4.8 satisfy all the requirements of random number distribution [26.4.1.5]. Descriptions are provided here only for operations on the distributions that are not described in those requirements or for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment operators, and for equality and inequality operators are not shown in the synopses.

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

3 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.

26.4.8.1 Uniform distributions [lib.rand.dist.uni]

26.4.8.1.1 Class template `uniform_int_distribution` [lib.rand.dist.uni.int]

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

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

```
template <class IntType = int>
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 <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
  template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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:* $a \le b$.

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.

26.4.8.1.2 Class template `uniform_real_distribution` [lib.rand.dist.uni.real]

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

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

```
  template <class RealType = double>
  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 <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, 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:* $\mathtt{a} \le \mathtt{b}$.

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.

26.4.8.2 Bernoulli distributions [lib.rand.dist.bern]

26.4.8.2.1 Class `bernoulli_distribution` [lib.rand.dist.bern.bernoulli]

1 A `bernoulli_distribution` random number distribution produces `bool` values b distributed according to the discrete probability function

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

```
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 <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
  template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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:* $0 \le$ `p` ≤ 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.

26.4.8.2.2 Class template `binomial_distribution` [lib.rand.dist.bern.bin]

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

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

```
  template <class IntType = int>
  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 <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, 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.

26.4.8.2.3 Class template `geometric_distribution` [lib.rand.dist.bern.geo]

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

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

```
  template <class IntType = int>
  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 <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, 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.

26.4.8.2.4 Class template `negative_binomial_distribution` [lib.rand.dist.bern.negbin]

1 A `negative_binomial_distribution` random number distribution produces random integers $i \geq 0$ distributed according to the discrete probability function

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

```
template <class IntType = int>
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 <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
  template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

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

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

2 *Requires:* $0 <$ `p` ≤ 1 and $0 <$ `k`.

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

`IntType k() const;`

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

`double p() const;`

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

26.4.8.3 Poisson distributions [lib.rand.dist.pois]

26.4.8.3.1 Class template `poisson_distribution` [lib.rand.dist.pois.poisson]

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

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

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

```
template <class IntType = int>
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 <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
  template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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);`

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

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

`double mean() const;`

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

26.4.8.3.2 Class template `exponential_distribution` [lib.rand.dist.pois.exp]

1 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 <class RealType = double>
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 <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
  template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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.

26.4.8.3.3 Class template `gamma_distribution` [lib.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 <class RealType = double>
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 <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
  template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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.

26.4.8.3.4 Class template `weibull_distribution` [lib.rand.dist.pois.weibull]

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

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

```
template <class RealType = double>
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 <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
  template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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.

26.4.8.3.5 Class template `extreme_value_distribution` [lib.rand.dist.pois.extreme]

1 An `extreme_value_distribution` random number distribution produces random numbers x distributed according to

the probability density function[8)]

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

```
template <class RealType = double>
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 <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
  template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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:* $0 <$ `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.

[8)] 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.

26.4.8.4 Normal distributions [lib.rand.dist.norm]

26.4.8.4.1 Class template `normal_distribution` [lib.rand.dist.norm.normal]

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

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

The distribution parameters μ and σ are also known as this distribution's *mean* and *standard deviation*.

```
template <class RealType = double>
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 <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
  template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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.

26.4.8.4.2 Class template `lognormal_distribution` [lib.rand.dist.norm.lognormal]

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

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

```
template <class RealType = double>
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 <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
  template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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:* $0 <$ `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.

26.4.8.4.3 Class template `chi_squared_distribution` [lib.rand.dist.norm.chisq]

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

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

where n is a positive integer.

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

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

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

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

`explicit chi_squared_distribution(int n = 1);`

2 *Requires:* $0 <$ n.

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

`int n() const;`

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

26.4.8.4.4 Class template `cauchy_distribution` [lib.rand.dist.norm.cauchy]

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

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

```
template <class RealType = double>
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 <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
  template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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:* $0 <$ `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.

26.4.8.4.5 Class template `fisher_f_distribution` [lib.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

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

where m and n are positive integers.

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

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

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

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

`explicit fisher_f_distribution(int m = 1, int 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.

`int m() const;`

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

`int n() const;`

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

26.4.8.4.6 Class template `student_t_distribution` [lib.rand.dist.norm.t]

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

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

where *n* is a positive integer.

```
template <class RealType = double>
class student_t_distribution
{
public:
  // types
  typedef RealType result_type;
  typedef unspecified param_type;

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

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

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

`explicit student_t_distribution(int n = 1);`

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

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

`int n() const;`

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

26.4.8.5 Sampling distributions [lib.rand.dist.samp]

26.4.8.5.1 Class template `discrete_distribution` [lib.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

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

```
template <class IntType = int>
class discrete_distribution
{
public:
  // types
  typedef IntType result_type;
  typedef unspecified param_type;

  // constructor and reset functions
  discrete_distribution();
  template <class InputIterator>
    discrete_distribution(InputIterator firstW, InputIterator lastW);
  explicit discrete_distribution(const param_type& parm);
  void reset();

  // generating functions
  template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
  template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, 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();`

2 *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>
discrete_distribution(InputIterator firstW, InputIterator lastW);
```

3 *Requires:*

a) `InputIterator` shall satisfy the requirements of an input iterator [lib.input.iterator].

b) If `firstW == lastW`, let the sequence w have length $n = 1$ and consist of the single value $w_0 = 1$. Otherwise, `[firstW,lastW)` shall form a sequence w of length $n > 0$ and `*firstW` shall yield a value w_0 convertible to `double`. [*Note:* The values w_k are commonly known as the *weights*. — *end note*]

c) The following relations shall hold: $w_k \geq 0$ for $k = 0, \ldots, n-1$, and $0 < S = w_0 + \cdots + w_{n-1}$.

4 *Effects:* Constructs a `discrete_distribution` object with probabilities

$$p_k = \frac{w_k}{S} \text{ for } k = 0, \ldots, n-1.$$

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

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

26.4.8.5.2 Class template `piecewise_constant_distribution` [lib.rand.dist.samp.pconst]

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

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

The $n+1$ distribution parameters b_i are also known as this distribution's *interval boundaries*.

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

  // constructor and reset functions
  piecewise_constant_distribution();
  template <class InputIteratorB, class InputIteratorW>
    piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                    InputIteratorW firstW);
  explicit piecewise_constant_distribution(const param_type& parm);
  void reset();

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

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

```
    result_type min() const;
    result_type max() const;
  };
```

```
piecewise_constant_distribution();
```

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

```
template <class InputIteratorB, class InputIteratorW>
piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB, InputIteratorW firstW);
```

3 *Requires:*

a) `InputIteratorB` shall satisfy the requirements of an input iterator [lib.input.iterator], as shall `InputIteratorW`.

b) If `firstB == lastB`,

(a) let the sequence w ~~shall~~ have length $n = 1$ and consist of the single value $w_0 = 1$, and

(b) let the sequence b ~~shall~~ have length $n+1$ with $b_0 = 0$ and $b_1 = 1$.

Otherwise,

(c) `[firstB, lastB)` shall form a sequence b of length $n+1$ whose leading element b_0 shall be convertible to `result_type`, and

(d) the length of the sequence w starting from `firstW` shall be at least n, `*firstW` shall return a value w_0 that is convertible to `double`, and any w_k for $k \geq n$ shall be ignored by the distribution.

[*Note:* The values w_k are commonly known as the *weights*. —*end note*]

c) The following relations shall hold for $k = 0, \ldots, n-1$: $b_k < b_{k+1}$ and $0 \leq w_k$. Also, $0 < S = w_0 + \cdots + w_{n-1}$.

4 *Effects:* Constructs a `piecewise_constant_distribution` object with probability densities

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

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

5 *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;
```

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

26.4.8.5.3 Class template `general_pdf_distribution` [lib.rand.dist.samp.genpdf]

1 A `general_pdf_distribution` random number distribution produces random numbers x, $x_{\min} \leq x < x_{\max}$, distributed

according to the probability density function

$$p(x\,|\,x_{\min}, x_{\max}, \rho) = \rho(x)\ \text{, for } x_{\min} \le x < x_{\min}\ .$$

```
template <class RealType = double>
class general_pdf_distribution
{
public:
  // types
  typedef RealType result_type;
  typedef unspecified param_type;

  // constructor and reset functions
  general_pdf_distribution();
  template <class Func>
    general_pdf_distribution(result_type xmin, result_type xmax, Func pdf);
  explicit general_pdf_distribution(const param_type& parm);
  void reset();

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

  // property functions
  result_type xmin() const;
  result_type xmax() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
};
```

```
general_pdf_distribution();
```

2 *Effects:* Constructs a `general_pdf_distribution` object with $x_{min} = 0$ and $x_{max} = 1$ such that $p(x) = 1$ for all $x_{min} \le x < x_{max}$.

```
template <class Func>
general_pdf_distribution(result_type xmin, result_type xmax, Func pdf);
```

3 *Requires:*

a) `pdf` shall be callable with one argument of type `result_type`, and shall return values of a type convertible to `double`;

b) $x_{\min} < x_{\max}$, and for all $x_{\min} \le x < x_{\max}$, `pdf`$(x)$ shall return a value that is non-negative, non-NaN, and non-infinity; and

c) the following relations shall hold:

$$0 < z = \int_{x_{\min}}^{x_{\max}} f(x)\,dx < \infty\,,$$

where f is the mathematical function corresponding to the supplied `pdf`. [*Note:* This implies that the user-supplied `pdf` need not be normalized. —*end note*]

4 *Effects:* Constructs a `general_pdf_distribution` object; `xmin` and `xmax` correspond to the respective parameters of the distribution and the corresponding probability density function is given by $\rho(x) = f(x)/z$.

```
result_type xmin() const;
```

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

```
result_type xmax() const;
```

6 *Returns:* The value of the `xmax` parameter with which the object was constructed.

Index