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

Document number:	WG21/N1932= J16/06-0002
Date:	2006-02-23
Revises:	None
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, IL 60510-0500 U.S.A.

This document proposes the incorporation into C++0X of a facility for random number generation. Complete proposed wording is provided, together with proposed context from N1905 = Becker: Working Draft.

The proposal is termed "comprehensive" because its contents consist of an edited amalgamation of all relevant earlier WG21/J16 papers and reflector messages that deal, in whole or in part, with this topic. These papers include:

N1398 = Maurer: A Proposal to Add an Extensible Random Number Facility to the Standard Library

N1588 = Paterno: On Random-Number Distributions for C++0X

N1836 = Austern: Draft Technical Report on C++ Library Extensions

N1837 = Austern: Library Extension Technical Report—Issues List Revision 10

N1914 = Paterno, *et al.*: A Proposal to Add Random-Number Distributions to C++0x

N1926 = Hinnant: C++ Standard Library Active Issues List (Revision R40)

Along the way, we have remedied numerous typographical oversights from these papers[1)] and have made a number of editorial adjustments[2)]. Additionally, we have taken the liberty of incorporating (1) proposed resolutions to all the issues raised to date, as well as (2) a few new elements based on our experiences in implementing the TR1 facilities as well as the proposed extensions thereto. Explanations and rationale for these are provided in a companion paper, N1933 = Brown, *et al.*: Improvements to TR1's Facility for Random Number Generation.

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.

---

1)Example: appending semicolons to the end of declarations lacking them.

2)Example: incorporating the `typedef`s from the "Engines with predefined parameters" section into the Synopsis section.

Contents

List of Tables

26 Numerical facilities [lib.numerical]

1 This clause descibes components that C++ programs may use to perform numerical and seminumerical operations.

2 The following subclauses describe components for complex number types, random number generation, numeric (*n*-at-a-time) arrays, generalized numeric algorithms, and facilities included from the ISO C library, as summarized in Table 1.

Table 1: Numerical library summary

Subclause	Header(s)
26.1 Numeric type requirements	
26.2 Complex numbers	<complex>
26.3 Random number generation	<random>
26.4 Numeric arrays	<valarray>
26.5 Generalized numeric operations	<numeric>
26.6 C library	<cmath> <math.h>

26.1 Numeric type requirements [lib.numeric.requirements]

1 . . .

26.2 Complex numbers [lib.complex.numbers]

1 . . .

26.3 Random number generation [lib.random.numbers]

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

2 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 `std::tr1::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 which result is always taken modulo 2^w.

26.3.1 Requirements [lib.rand.req]

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

1 The effect of instantiating a template

a) that has a template type parameter named `UniformRandomNumberGenerator` is undefined unless that type satisfies the requirements of uniform random number generator [26.3.1.2].

b) that has a template type parameter named `Engine` is undefined unless that type satisfies the requirements of uniform random number engine [26.3.1.3].

c) that has a template type parameter named `RealType` is undefined unless that type is one of `float`, `double`, or `long double`.

d) that has a template type parameter named `IntType` is undefined unless that type is one of `short`, `int`, `long`, `unsigned short`, `unsigned int`, or `unsigned long`.[3)]

e) that has a template type parameter named `UIntType` is undefined unless that type is one of `unsigned short`, `unsigned int`, or `unsigned long`.[4)]

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.

26.3.1.2 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 2 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`.

[3)] It is intended that this list be augmented with `long long` and `unsigned long long` if and when these types are incorporated into the Working Paper.

[4)] It is intended that this list be augmented with `unsigned long long` if and when this type is incorporated into the Working Paper.

Table 2: Uniform random number generator requirements

expression	return type	pre/post-condition	complexity
`X::result_type`	`T`	`T` is an arithmetic type [basic.fundamental] other than `bool`.	compile-time
`u()`	`T`	If `X` is integral, 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, 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, denotes the greatest value potentially returned by `operator()`; otherwise denotes 1.	compile-time

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

1 A class `X` satisfies the requirements of a random number engine if the expressions shown in table 3 are valid and have the indicated semantics, and if `X` also satisfies all other requirements of this section 26.3.1.3. In that table and throughout this section 26.3.1.3,

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 type;

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

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

g) `is` is an lvalue of the type of some class template specialization `basic_istream<charT, traits>`;

where `charT` and `traits` are constrained according to [lib.strings] and [lib.input.output].

2 A random number engine `x` has at any given time a state $\mathtt{x}_i$ for some integer $i \geq 0$. Upon successful instantiation, a random number engine `x` has an initial state $\mathtt{x}_0$. An engine's state may be established by invoking its 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 $\mathtt{x}_i$ is advanced to its *successor state* $\mathtt{x}_{i+1}$, and

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

Table 3: 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(s)`	—	Creates an engine with initial state determined by `static_cast<unsigned long>(s)`.	$\mathcal{O}$(size of state)
`X(g)`	—	Creates an engine with initial state determined by the results of successive invocations of `g`.	$\mathcal{O}$(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)`	`void`	post: If `g` does not throw, `u == X(g)`. 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)`
`u()`	`T`	Sets the state to $\mathtt{u}_{i+1} = \mathsf{TA}(\mathtt{u}_i)$ and returns $\mathsf{GA}(\mathtt{u}_i)$.	amortized constant
`x == y`	`bool`	With S_x and S_y as the infinite sequences of values that would be generated by repeated calls to `x()` and `y()`, respectively, returns `true` if $S_x = S_y$; returns `false` otherwise.	$\mathcal{O}$(size of state)
`x != y`	`bool`	`!(x == y)`	$\mathcal{O}$(size of state)
`os << x`	reference to the type of `os`	With `os.`*fmtflags* set to `ios_base::dec|ios_base::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.	$\mathcal{O}$(size of state)

expression	return type	pre/post-condition	complexity
`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, 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.	$\mathcal{O}$(size of state)

4 `X` shall satisfy the requirements of uniform random number generator [26.3.1.2] as well as of `CopyConstructible` [lib.copyconstructible] and of `Assignable` [lib.container.requirements]. Copy construction and assignment shall each be of complexity $\mathcal{O}$(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`.

26.3.1.4 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 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 4 are valid and have the indicated semantics, and if `X` and its associated types also satisfies all other requirements of this section 26.3.1.4. In that table and throughout this section,

a) $\mathtt{B}_i$ is the type of the i^{th} of `X`'s base engines; 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 4: Random number engine adaptor requirements

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

3 X shall satisfy the requirements of random number engine [26.3.1.3], subject to the following interpretations of those requirements.

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

f) The equality operator, applied to two operands `a1` and `a2` of identical engine adaptor type, returns true if and only if each base engine of `a1` compares equal, in turn, to the corresponding base engine of `a2`.

4 X shall have one additional constructor with as many arguments as X has base engines. These arguments' types shall correspond to the types of the base engines. The constructor shall construct X, initializing each of its base engines, in turn, with a copy of the value of the corresponding argument, in no way modifying any of the argument values.

26.3.1.5 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 5 are valid and have the indicated semantics, and if X and its associated types also satisfies all other requirements of this section 26.3.1.5. 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` and `v` are values 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 [26.3.1.2];

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 identified as the *parameters of the distribution*.

Table 5: 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`	`P` is constructible from the identical values used in the construction of any value of `X`.	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
`u.param()`	`P`	Returns a value that could have been used to construct `u` in its initial state.	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`	The sequence of numbers returned by successive invocations with the same object `e` is randomly distributed according to the associated $p(z)$ or $P(z_i)$ function.	amortized constant number of invocations of `e`
`u(e,p)`	`T`	Returns the same result as would `v(e)` where `v` had been freshly constructed with argument `p`.	No worse than the complexity of `X v(p); v(e);`.
`x.min()`	`T`	Returns `glb`.	constant
`x.max()`	`T`	Returns `lub`.	constant

expression	return type	pre/post-condition	complexity
`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.	$\mathscr{O}$(size of state)
`is >> u`	reference to the type of `is`	Restores from `is` the parameters and additional internal data of `u`. If bad input is encountered, 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.	$\mathscr{O}$(size of state)

3 `X` shall satisfy the requirements of `CopyConstructible` [lib.copyconstructible] and `Assignable` [lib.container.requirements]. Copy construction and assignment shall each be of complexity $\mathscr{O}$(size of state).

4 The sequence of numbers produced by repeated invocations of `x(e)` shall be independent of any invocation of `os << 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 26.3 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]; Copy construction and assignment shall each be of complexity $\mathscr{O}$(number of values used in the source's construction).

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.

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

```
namespace std {
  // [26.3.3.1] Class template linear_congruential_engine
```

```
template <class UIntType, UIntType a, UIntType c, UIntType m>
  class linear_congruential_engine;

// [26.3.3.2] Class template mersenne_twister_engine
template <class UIntType, int w, int n, int m, int r,
          UIntType a, int u, int s, UIntType b, int t, UIntType c, int l>
  class mersenne_twister_engine;

// [26.3.3.3] Class template subtract_with_carry
template <class IntType, IntType m, int s, int r>
  class subtract_with_carry_engine;

// [26.3.3.4] Class template subtract_with_carry_01_engine
template <class RealType, int w, int s, int r>
  class subtract_with_carry_01_engine;

// [26.3.4.1] Class template discard_block_engine
template <class Engine, int p, int r>
  class discard_block_engine;

// [26.3.4.2] Class template shuffle_order_engine
template <class Engine, int k>
  class shuffle_order_engine;

// [26.3.4.3] Class template xor_combine_engine
template <class Engine1, int s1, class Engine2, int s2>
  class xor_combine_engine;

// [26.3.5] Engines with predefined parameters
typedef see below minstd_rand0;
typedef see below minstd_rand;
typedef see below mt19937;
typedef see below ranlux_base_01;
typedef see below ranlux64_base_01;
typedef see below ranlux3;
typedef see below ranlux4;
typedef see below ranlux3_01;
typedef see below ranlux4_01;
typedef see below knuth_b;

// [26.3.6] Class random_device
class random_device;

// [26.3.7.1.1] Class template uniform_int_distribution
template <class IntType = int>
  class uniform_int_distribution;

// [26.3.7.1.2] Class template uniform_real_distribution
template <class RealType = double>
  class uniform_real_distribution;
```

```
// [26.3.7.2.1] Class bernoulli_distribution
class bernoulli_distribution;

// [26.3.7.2.2] Class template binomial_distribution
template <class IntType = int>
  class binomial_distribution;

// [26.3.7.2.3] Class template geometric_distribution
template <class IntType = int>
  class geometric_distribution;

// [26.3.7.2.4] Class template negative_binomial_distribution
template <class IntType = int>
  class negative_binomial_distribution;

// [26.3.7.3.1] Class template poisson_distribution
template <class IntType = int>
  class poisson_distribution;

// [26.3.7.3.2] Class template exponential_distribution
template <class RealType = double>
  class exponential_distribution;

// [26.3.7.3.3] Class template gamma_distribution
template <class RealType = double>
  class gamma_distribution;

// [26.3.7.3.4] Class template weibull_distribution
template <class RealType = double>
  class weibull_distribution;

// [26.3.7.3.5] Class template extreme_value_distribution
template <class RealType = double>
  class extreme_value_distribution;

// [26.3.7.4.1] Class template normal_distribution
template <class RealType = double>
  class normal_distribution;

// [26.3.7.4.2] Class template lognormal_distribution
template <class RealType = double>
  class lognormal_distbution;

// [26.3.7.4.3] Class template chi_squared_distribution
template <class RealType = double>
  class chi_squared_distribution;

// [26.3.7.4.4] Class template cauchy_distribution
template <class RealType = double>
```

```
    class cauchy_distribution;

  // [26.3.7.4.5] Class template fisher_f_distribution
  template <class RealType = double>
    class fisher_f_distribution;

  // [26.3.7.4.6] Class template student_t_distribution
  template <class RealType = double>
    class student_t_distribution;

  // [26.3.7.5.1] Class template discrete_distribution
  template <class IntType = int>
    class discrete_distribution;

  // [26.3.7.5.2] Class template piecewise_constant_distribution
  template <class RealType = double>
    class piecewise_constant_distribution;

  // [26.3.7.5.3] Class template general_pdf_distribution
  template <class RealType = double>
    class general_pdf_distribution;
} // namespace std
```

26.3.3 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 for constructors and for the `seed` functions that take a zero-argument function object, no function described in this section 26.3.3 throws an exception.

3 Except where specified otherwise, the class templates specified in this section 26.3.3 satisfy the requirements of random number engine [26.3.1.3]. 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, and for equality and inequality operators are not shown in the synopsis.

26.3.3.1 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 multiplier = a;
  static const UIntType increment = c;
```

```
    static const UIntType modulus = m;
    static const result_type min = c == 0u ? 1u: 0u;
    static const result_type max = m - 1u ;

    // constructors and seeding functions
    explicit linear_congruential_engine(unsigned long s = 1uL);
    template <class Gen> explicit linear_congruential_engine(Gen& g);
    void seed(unsigned long s = 1uL);
    template <class Gen> void seed(Gen& g);

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

2 The template parameter `UIntType` shall denote an unsigned integral type large enough to store values as large as $m-1$. If the template parameter `m` is 0, the modulus m used throughout this section 26.3.3.1 is `numeric_limits<UIntType>::max()` plus 1. [*Note:* The result is not representable as a value of type `UIntType`. —*end note*] Otherwise, the following relations shall hold: `a` $<$ `m` and `c` $<$ `m`.

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

`linear_congruential_engine(unsigned long s = 1uL);`

4 *Effects:* Constructs a `linear_congruential_engine` object as if by invoking `seed(s)`.

`void seed(unsigned long s = 1uL);`

5 *Effects:* If c mod m is 0 and `s` mod m is 0, sets the engine's state to 1, otherwise sets the engine's state to `s` mod m.

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

6 *Effects:* If c mod m is 0 and `g()` mod m is 0, sets the engine's state to 1, else sets the engine's state to `g()` mod m.

7 *Complexity:* Exactly one invocation of g.

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

1 A `mersenne_twister_engine` random number engine[5)] 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 $X_{i+m-n} \,\mathsf{xor}\, (Y \,\mathsf{rshift}\, 1) \,\mathsf{xor}\, \alpha$.

5) 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.

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

  // parameter values and engine characteristics
  static const int word_size = w;
  static const int state_size = n;
  static const int shift_size = m;
  static const int mask_bits = r;
  static const UIntType xor_mask = a;
  static const int tempering_u = u;
  static const int tempering_s = s;
  static const UIntType tempering_b = b;
  static const int tempering_t = t;
  static const UIntType tempering_c = c;
  static const int tempering_l = l;
  static const result_type min = 0;
  static const result_type max = 2^w - 1;

  // constructors and seeding functions
  explicit mersenne_twister_engine(unsigned long value = 5489uL);
  template <class Gen> explicit mersenne_twister_engine(Gen& g);
  void seed(unsigned long value = 5489uL);
  template <class Gen> void seed(Gen& g);

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

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: $1 \leq m \leq n$. $0 \leq r, u, s, t, l \leq w$. $0 \leq a, b, c \leq 2^w - 1$.

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

```
mersenne_twister_engine(unsigned long value = 5489uL);
```

6 *Effects:* Constructs a `mersenne_twister_engine` object as if by invoking `seed(value)`.

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

7 *Effects:* Given the values $z_0, \ldots, z_{n-1}$ obtained by successive invocations of g, sets $X_{-n}, \ldots, X_{-1}$ to z_0 mod 2^w, $\ldots, z_{n-1}$ mod 2^w, respectively.

8 *Complexity:* Exactly n invocations of g.

```
void seed(unsigned long value);
```

9 *Effects:* Sets X_{-n} to value mod 2^w. Then, iteratively for $i = 1-n, \ldots, -1$, sets X_{i-n} to

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

10 *Complexity:* $\mathscr{O}(\mathrm{n})$.

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

1 A `subtract_with_carry_engine` random number engine produces unsigned integer random numbers. The state $\mathtt{x}_i$ of a `subtract_with_carry_engine` object x is of size $r+1$. and consists of a sequence X of r integer values $0 \le X_i < m$; all subscripts applied to X are to be taken modulo r. The state $\mathtt{x}_i$ additionally consists of an integer c (known as the *carry*) whose value is either 0 or 1.

2 The transition algorithm is a modular linear function of the form $\mathsf{TA}(\mathtt{x}_i) = (a \cdot \mathtt{x}_i) \bmod p$, where p is of the form $m^r - m^s + 1$ 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 mod m. Set c to 1 if $Y < 0$, otherwise set c to 0.

3 The generation algorithm yields the last value of Y mod m produced as a result of advancing the engine's state as described above.

```
template <class IntType, IntType m, int s, int r>
class subtract_with_carry
{
public:
  // types
  typedef IntType result_type;

  // parameter values and engine characteristics
  static const IntType modulus = m;
  static const int short_lag = s;
  static const int long_lag = r;
  static const result_type min = 0;
  static const result_type max = m - 1;

  // constructors and seeding functions
  explicit subtract_with_carry_engine(unsigned long value = 19780503uL);
  template <class Gen> explicit subtract_with_carry_engine(Gen& g);
  void seed(unsigned long value = 19780503uL);
  template <class Gen> void seed(Gen& g);

  // generating functions
```

```
    result_type operator()();
  };
```

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

5 The following relation shall hold: $0 < \texttt{s} < \texttt{r}$.

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

```
subtract_with_carry_engine(unsigned long value = 19780503uL);
```

7 *Effects:* Constructs a `subtract_with_carry_engine` object as if by invoking `seed(value)`.

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

8 *Effects:* With $n = \lfloor (m+31)/32 \rfloor$ and given the values $z_0, \ldots, z_{n*r-1}$ obtained by successive invocations of `g`, sets $X_{-r}, \ldots, X_{-1}$ to $(z_0 \cdot 2^{32} + \cdots + z_{n-1} \cdot 2^{32(n-1)}) \bmod m, \ldots, (z_{(r-1)n} \cdot 2^{32} + \cdots + z_{r-1} \cdot 2^{32(n-1)}) \bmod m$, respectively. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.

9 *Complexity:* Exactly `r*n` invocations of `g`.

```
void seed(unsigned long value = 19780503uL);
```

10 *Effects:* If `value` is 0, sets `value` to 19780503. In any case, with a linear congruential generator $\mathsf{lcg}(i)$ having parameters $m_{\mathsf{lcg}} = 2147483563$, $a_{\mathsf{lcg}} = 40014$, $c_{\mathsf{lcg}} = 0$, and $\mathsf{lcg}(0) = \texttt{value}$, sets $X_{-r}, \ldots, X_{-1}$ to $\mathsf{lcg}(1) \bmod m, \ldots, \mathsf{lcg}(r) \bmod m$, respectively. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.

11 *Complexity:* $\mathscr{O}(\mathrm{r})$.

26.3.3.4 Class template `subtract_with_carry_01_engine` [lib.rand.eng.sub1]

1 A `subtract_with_carry_01_engine` random number engine produces floating-point random numbers. The state $\mathtt{x}_i$ of a `subtract_with_carry_01_engine` object `x` is of size $\mathscr{O}(\mathrm{r})$, 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 $\mathtt{x}_i$ 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 $\mathsf{TA}(\mathtt{x}_i) = (a \cdot \mathtt{x}_i) \bmod p$, 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` [26.3.3.3] with $m = 2^w$. — *end note*]

3 The generation algorithm is $\mathsf{GA}(\mathtt{x}_i) = T \cdot 2^{-w} + \varepsilon$ 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;

  // constructors and seeding functions
  explicit subtract_with_carry_01_engine(unsigned long value = 19780503uL);
  template <class Gen> explicit subtract_with_carry_01_engine(Gen& g);
  void seed(unsigned long value = 19780503uL);
  template <class Gen> void seed(Gen& g);

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

4 The following relations shall hold: $0 < \mathtt{s} < \mathtt{r}$ and $\mathtt{w} <$ `numeric_limits<RealType>::min_exponent`. [*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 $n = \lfloor(\mathtt{w}+31)/32\rfloor$ 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*]

```
subtract_with_carry_01_engine(unsigned long value = 19780503uL);
```

6 *Effects:* Constructs a `subtract_with_carry_01_engine` object as if by invoking `seed(value)`.

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

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

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

```
void seed(unsigned long value = 19780503uL);
```

9 *Effects:* If `value` is 0, sets `value` to 19780503uL. In any case, with a linear congruential generator lcg(i) having parameters $m_{lcg} = 2147483563$, $a_{lcg} = 40014$, $c_{lcg} = 0$, and $lcg(0) =$ `value`, sets carry(-1) and x(-r), …, x(-1) to the values that would be set by calling `seed(lcg)`. as described above.

10 *Complexity:* $\mathscr{O}(n \cdot \mathtt{r})$.

26.3.4 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 for constructors and for the `seed` functions that take a zero-argument function object, no function described in this section 26.3.4 throws an exception.

3 Except where specified otherwise, the class templates specified in this section satisfy the requirements of random number engine [26.3.1.3]. 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.

26.3.4.1 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 b. The state $\mathtt{x}_i$ of a `discard_block_engine` engine adaptor object `x` consists of the state $\mathtt{b}_i$ of its base engine `b` and an additional integer n. The size of the state is the size of b's state plus 1.

2 The transition algorithm discards all but $r > 0$ values from each block of $p \geq r$ values delivered by b. The state transition is performed as follows: If $n \geq r$, advance the state of `b` from $\mathtt{b}_i$ to $\mathtt{b}_{i+p-r}$ and set n to 0. In any case, then increment n and advance `b`'s then-current state $\mathtt{b}_j$ to $\mathtt{b}_{j+1}$.

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

```
template <class Engine, int p, int 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 block_size = p;
  static const int used_block = r;
  static const auto min = base_type::min;
  static const auto 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 s);
  template <class Gen> explicit discard_block_engine(Gen& g);
  void seed();
  void seed(unsigned long s);
  template <class Gen> void seed(Gen& g);

  // generating functions
  result_type operator()();

  // property functions
```

```
    const base_type& base() const;

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

4 The following relation shall hold: $1 \leq$ `r` $\leq$ `p`.

5 The textual representation consists of the textual representation of `b` followed by the value of `n`.

`discard_block_engine();`

6 *Effects:* Constructs a `discard_block_engine` object. To construct the subobject `b`, invokes the default constructor of `base_type`. Sets `n` to 0.

`discard_block_engine(const base_type& urng);`

7 *Effects:* Constructs a `discard_block_engine` object. Initializes `b` with a copy of `urng`. Sets `n` to 0.

`discard_block_engine(unsigned long s);`

8 *Effects:* Constructs a `discard_block_engine` object. To construct the subobject `b`, invokes the `base_type(s)` constructor. Sets `n` to 0.

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

9 *Effects:* Constructs a `discard_block_engine` object. To construct the subobject `b`, invokes the `base_type(g)` constructor. Sets `n` to 0.

`void seed();`

10 *Effects:* Invokes `b.seed()` and sets `n` to 0.

`void seed(unsigned long s);`

11 *Effects:* Invokes `b.seed(s)` and sets `n` to 0.

`const base_type& base() const;`

12 *Returns:* A reference to `b`.

26.3.4.2 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 b, 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 $\mathtt{b}_i$ of its base engine `b`, an additional value Y of the type delivered by `b`, and an additional sequence V of k values also of the type delivered by `b`. The size of the state is the size of b's state plus $k+1$.

2 The transition algorithm permutes the values produced by b. 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 b is integer-valued, or as $\lfloor k \cdot Y \rfloor$, if b 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 `b`'s state as described above.

```
template <class Engine, int 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 table_size = k;
  static const auto min = base_type::min;
  static const auto 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 s);
  template <class Gen> explicit shuffle_order_engine(Gen& g);
  void seed();
  void seed(unsigned long s);
  template <class Gen> void seed(Gen& g);

  // generating functions
  result_type operator()();

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

private:
  base_type   b;          // 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 `b`, followed by the `k` values of V, followed by the value of Y.

`shuffle_order_engine();`

6 *Effects:* Constructs a `shuffle_order_engine` object. To construct the subobject `b`, invokes the default constructor of `base_type`. Initializes `V[0]`, ..., `V[k-1]` and Y, in that order, with values obtained from successive invocations of `b()`.

`shuffle_order_engine(const base_type& urng);`

7 *Effects:* Constructs a `shuffle_order_engine` object. Initializes `b` with a copy of `urng`. Initializes `V[0]`, ..., `V[k-1]` and Y as described above.

```
shuffle_order_engine(unsigned long s);
```

8 *Effects:* Constructs a `shuffle_order_engine` object. To construct the subobject `b`, invokes the `base_type(s)` constructor. Initializes `V[0]`, ..., `V[k-1]` and Y as described above.

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

9 *Effects:* Constructs a `shuffle_order_engine` object. To construct the subobject `b`, invokes the `base_type(g)` constructor. Initializes `V[0]`,
;*ldots*, `V[k-1]` and Y as described above.

```
void seed();
```

10 *Effects:* Invokes `b.seed()` and initializes `V[0]`, ..., `V[k-1]` and Y, as described above.

```
void seed(unsigned long s);
```

11 *Effects:* Invokes `b.seed(s)` and initializes `V[0]`, ..., `V[k-1]` and Y, as described above.

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

12 *Returns:* A reference to `b`.

26.3.4.3 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 `b1` and `b2` by merging their left-shifted random values via bitwise exclusive-or. The state $\mathtt{x}_i$ of a `xor_combine_engine` engine adaptor object `x` consists of the states $\mathtt{b1}_i$ and $\mathtt{b2}_i$ of its base engines. The size of the state is the size of the state of `b1` plus the size of the state of `b2`.

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

3 The generation algorithm is $\mathsf{GA}(\mathtt{x}_i) = (\mathtt{b1()}\ \mathsf{lshift}_w\, \mathtt{s1})\ \mathsf{xor}\ (\mathtt{b2()}\ \mathsf{lshift}_w\, \mathtt{s2})$, where w denotes the value of `numeric_limits<result_type>::digits`.

```
template <class Engine1, int s1, class Engine2, int s2>
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 shift1 = s1;
  static const int 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 s);
    template <class Gen> explicit xor_combine_engine(Gen& g);
    void seed();
    template <class Gen> void seed(Gen& g);

    // generating functions
    result_type operator()();

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

  private:
    base1_type b1;               // exposition only
    base2_type b2;               // exposition only
  };
```

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

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 be nonzero. (If both `s1` and `s2` are nonzero then the low bits will always be zero.)

b) It is also advisable for the unshifted base engine's `max` to be $2^n - 1 - \texttt{min}$ for some non-negative integer n, and for the shift applied to the other base engine 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) $m_1 = (\texttt{Engine1::max} - \texttt{Engine1::min})\ \mathsf{lshift}_w\, (\texttt{s1} - \texttt{s2})$,

b) $m_2 = \texttt{Engine2::max} - \texttt{Engine2::min}$,

c) $A = m_1\ \mathsf{bitand}\ m_2$,

d) $B = m_1\ \mathsf{bitor}\ m_2$, and

e) $C = 0$ if A is zero and $C = 2^{\log_2 A} - 1$ if A is nonzero,

the value of the member `max` is $(C\ \mathsf{bitor}\ B)\ \mathsf{lshift}_w\, (\texttt{s1} - \texttt{s2})$,

8 The textual representation consists of the textual representation of `b1` followed by the textual representation of `b2`.

```
xor_combine_engine();
```

9 *Effects:* Constructs a `xor_combine_engine` object. To construct each of the subobjects `b1` and `b2`, invokes their respective default constructors.

```
xor_combine_engine(const base1_type & urng1, const base2_type & urng2);
```

10 *Effects:* Constructs a `xor_combine_engine` object. Initializes the subobject `b1` with a copy of `urng1` and then initializes the subobject `b2` with a copy of `urng2`.

```
xor_combine_engine(unsigned long s);
```

11 *Effects:* Constructs a `xor_combine_engine` object. Initializes the subobject `b1` by invoking the `b1(s)` constructor, and then initializes the subobject `b2` by invoking the `b2(s+1)` constructor. [*Note:* If both `b1` and `b2` are of the same type, both base engines should not be initialized with the same seed. — *end note*]

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

12 *Effects:* Constructs a `xor_combine_engine` object. Initializes the subobject `b1` by invoking the `b1(g)` constructor, and then initializes the subobject `b2` by invoking the `b2(g)` constructor.

```
void seed();
```

13 *Effects:* Invokes `b1.seed()` and `b2.seed()`.

```
const base1_type& base1() const;
```

14 *Returns:* A reference to `b1`.

```
const base2_type& base2() const;
```

15 *Returns:* A reference to `b2`.

26.3.5 Engines with predefined parameters [lib.rand.predef]

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

1 *Requires:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `minstd_rand0` shall produce the value 1043618065.

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

2 *Requires:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `minstd_rand` shall produce the value 399268537.

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

3 *Requires:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `mt19937` shall produce the value 4123659995.

```
typedef subtract_with_carry_01_engine<float, 24, 10, 24>
        ranlux_base_01;
```

4 *Requires:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `ranlux_base_01` shall produce the value $7937952 \cdot 2^{-24}$.

```
typedef subtract_with_carry_01_engine<double, 48, 5, 12>
        ranlux64_base_01;
```

5 *Requires:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `ranlux64_base_01` shall produce the value $192113843633948 \cdot 2^{-48}$.

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

6 *Requires:* The 10000$^{\text{th}}$ 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 *Requires:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `ranlux4` shall produce the value 8587295.

```
typedef discard_block_engine<ranlux_base_01, 223, 24>
        ranlux3_01;
```

8 *Requires:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `ranlux3_01` shall produce the value $5957620 \cdot 2^{-24}$.

```
typedef discard_block_engine<ranlux_base_01, 389, 24>
        ranlux4_01;
```

9 *Requires:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `ranlux4_01` shall produce the value $8587295 \cdot 2^{-24}$.

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

10 *Requires:* The 10000$^{\text{th}}$ consecutive invocation of a default-constructed object of type `knuth_b` shall produce the value 1112339016.

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

1 A `random_device` produces non-deterministic random numbers. It satisfies the requirements of uniform random number generator [26.3.1.2]. 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 synopses.

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;

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

  // generating functions
  result_type operator()();

  // property functions
  result_type min() const;
  result_type max() const;
  double entropy() const;

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

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

3 *Effects:* Constructs a `random_device` non-deterministic random number engine. The semantics and default value of the `token` parameter are implementation-defined.[6]

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

`result_type min() const;`

5 *Returns:* `numeric_limits<result_type>::min()`.

`result_type max() const;`

6 *Returns:* `numeric_limits<result_type>::max()`.

`double entropy() const;`

7 *Returns:* An entropy estimate for the random numbers returned by operator(), in the range `min()` to $\log_2($`max()`$+ 1)$. [*Note:* A deterministic random number generator (*e.g.*, a random number engine) has entropy 0. — *end note*]

8 *Throws:* Nothing.

`result_type operator()();`

[6] The parameter is intended to allow an implementation to differentiate between different sources of randomness.

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

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

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

1 Except where specified otherwise, the classes and class templates specified in this section 26.3.7 satisfy all the requirements of random number distribution [26.3.1.5]. 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.

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

26.3.7.1.1 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 probability function

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

where a and b are the parameters of the distribution.

```
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;
```

```
};
```

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

2 *Requires:* a $\leq$ b.

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

```
result_type a() const;
```

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

```
result_type b() const;
```

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

26.3.7.1.2 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

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

where a and b are the parameters of the distribution.

```
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;
  };
```

```
uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);
```

2 *Requires:* $\mathtt{a} \leq \mathtt{b}$.

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.

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

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

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

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

where p is the parameter of the distribution.

```
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;
  };
```

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

2 *Requires:* $0 \leq \text{p} \leq 1$.

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.

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

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

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

where t and p are the parameters of the distribution.

```
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;
};
```

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

2 *Requires:* $0 \leq$ `p` ≤ 1 and $0 \leq$ `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.

26.3.7.2.3 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 probability function

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

where p is the parameter of the distribution.

```
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;
  };
```

```
geometric_distribution(double p = 0.5);
```

2 *Requires:* $0 < \texttt{p} < 1$.

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.

26.3.7.2.4 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 probability function

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

where k and p are the parameters of the distribution.

```
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 = 0, 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;
};
```

```
negative_binomial_distribution(IntType k = 0, double p = 0.5);
```

2 *Requires:* $0 \leq \texttt{p} \leq 1$ and $0 \leq \texttt{k}$.

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.

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

26.3.7.3.1 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 probability function

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

where μ, the *mean*, is the parameter of the distribution.

```
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;
};
```

```
poisson_distribution(double mean = 0.5);
```

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.

26.3.7.3.2 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

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

where λ is the parameter of the distribution.

```
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;
};
```

```
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.

26.3.7.3.3 Class template `gamma_distribution` [lib.rand.dist.pois.gamma]

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

$$p(x) = \frac{e^{-x}}{\Gamma(\alpha)} \cdot x^{\alpha-1} \ ,$$

where α is the parameter of the distribution.

```
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);
  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;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
};
```

`gamma_distribution(RealType alpha = 1.0);`

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

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

`RealType alpha() const;`

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

26.3.7.3.4 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

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

where a and b are the parameters of the distribution.

```
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;
};
```

`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.

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

1 An `extreme_value_distribution` random number distribution produces random numbers x distributed with probability density function

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

where a and b are the parameters of the distribution.

```
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;
};
```

`extreme_value_distribution(RealType a = 1.0, RealType b = 1.0);`

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

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

`RealType a() const;`

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

`RealType b() const;`

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

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

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

1 A `normal_distribution` random number distribution produces random numbers x distributed according to the proba-

bility density function

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-(x-\mu)^2/(2\sigma^2)} ,$$

where μ and σ, respectively the *mean* and the *standard deviation*, are the parameters of the distribution.

```
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;
};
```

`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.

26.3.7.4.2 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

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

where m and s are the parameters of the distribution.

```
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;
};
```

`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.

26.3.7.4.3 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

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

where n, a positive integer,is the parameter of the distribution.

```
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;
};
```

`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.

26.3.7.4.4 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

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

where a and b are the parameters of the distribution.

```
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;
};
```

`cauchy_distribution(RealType a = 0.0, RealType b = 1.0);`

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

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

`RealType a() const;`

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

`RealType b() const;`

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

26.3.7.4.5 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

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

where positive integers m and n are the parameters of the distribution.

```
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;
};
```

`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.

26.3.7.4.6 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

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

where integer n is the parameter of the distribution.

```
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;
};
```

`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.

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

26.3.7.5.1 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 probability function

$$P(i) = p_i \,,$$

where the n probabilities p_i are the parameters of the distribution.

```
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`, the sequence w shall have length $n = 1$ and shall 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 $k = 0, \ldots, n-1$.

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

1 A `piecewise_constant_distribution` random number distribution produces random numbers x, $x_0 \le x < x_n$, uniformly distributed over each subinterval $[x_i, x_{i+1})$ with probability

$$P(x_i \le x < x_{i+1}) = p_i \ ,$$

where the n probabilities p_i and the corresponding $n+1$ *interval boundaries* x_i are the parameters of the distribution.

```
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 InputIteratorX, InputIteratorD>
    piecewise_constant_distribution(InputIteratorX firstX, InputIteratorX lastX, InputIteratorD firstD);
  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> probabilities() 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 $n = 1$, $p_0 = 1$, $x_0 = 0$, and $x_1 = 1$.

```
template <class InputIteratorX, InputIteratorD>
piecewise_constant_distribution(InputIteratorX firstX, InputIteratorX lastX, InputIteratorD firstD);
```

3 *Requires:*

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

b) If `firstX == lastX`,

(a) the sequence ρ shall have length $n = 1$ and consist of the single value $\rho_0 = 1$, and

(b) the sequence x shall have length $n+1$ with $x_0 = 0$ and $x_1 = 1$.

Otherwise,

(c) `[firstX, lastX)` shall form a sequence x of length $n+1$ whose leading element x_0 shall be convertible to `result_type`, and

(d) the length of the sequence ρ starting from `firstD` shall be at least n, `*firstD` shall return a value ρ_0 that is convertible to `double`, and any ρ_k for $k \geq n$ shall be ignored by the distribution.

c) The following relations shall hold for $k = 0, \ldots, n-1$: $x_k < x_{k+1}$ and $0 \leq \rho_k$. Also, $0 < S = \rho_0 + \cdots + \rho_{n-1}$.

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

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

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

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

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

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

26.3.7.5.3 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 whose shape is determined when the distribution is constructed. $x_{\min}$ and $x_{\max}$ are parameters of the distribution.

```
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;
  };
```

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

2 *Requires:*

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

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

c) the following relations shall hold:

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

3 *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

$$p(x) = \frac{1}{z} \cdot \texttt{pdf}(x), \text{ where } x_{\min} \leq x < x_{\max}\,.$$

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

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

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

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

26.4 Numeric arrays [lib.numarray]

1 . . .

26.5 Generalized numeric operations [lib.numeric.ops]

1 . . .

26.6 C library [lib.c.math]

1 . . .

Index