Concepts for
Random Number Generation in C++0X

In anticipation of a National Body comment regarding the recently-approved Committee Draft, this document proposes adjustments to the description of the C++0X Standard Library’s random number facility in order to make full use of concepts as formulated in the following documents:

— N2773 = Gregor, et al.: Proposed Wording for Concepts (Revision 9),
— N2774 = Gregor, et al.: Foundational Concepts for the C++0X Standard Library (Revision 5),
— N2758 = Gregor, et al.: Iterator Concepts for the C++0X Standard Library (Revision 5),
— N2776 = Gregor, et al.: Concepts for the C++0X Standard Library: Containers (Revision 4),

We make every attempt to provide complete backward compatibility with the pre-concept Standard Library, and will note each area in which we have knowingly changed semantics.

This document is based on Clause 26.4 [random.numbers] of N2723 = Becker: Working Draft, Standard for Programming Language C++. Wherever the numbering of a (sub)section herein matches a (sub)section of that Working Draft, the text in this document should be considered replacement text, except that any relevant issue resolutions or editorial adjustments applied subsequent to that Draft will also need to be applied to this paper, perhaps with some small adjustments in wording.

Wherever possible within the proposed wording herein, adjustments to text will be denoted as added, deleted, or changed; editorial and similar remarks not part of the proposed wording will be shaded. Changes in the index are not specially marked.

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.
Contents

26 Numerics library

26.4 Random number generation ........................................... 3
  26.4.1 Requirements .................................................. 3
  26.4.2 Header <random> synopsis ..................................... 3
  26.4.3 Concepts and related requirements for random number generation ........................................... 7
    26.4.3.1 Concept UniformRandomNumberGenerator .................. 7
    26.4.3.2 Concept RandomNumberEngine .............................. 8
    26.4.3.3 Concept RandomNumberEngineAdaptor ..................... 10
    26.4.3.4 Concept RandomNumberDistribution ....................... 12
    26.4.3.5 Concept SeedSequence ................................. 14
  26.4.4 Random number engine class templates ......................... 16
    26.4.4.1 Class template linear_congruential_engine .............. 16
    26.4.4.2 Class template mersenne_twister_engine ............... 17
    26.4.4.3 Class template subtract_with_carry_engine ............. 19
  26.4.5 Random number engine adaptor class templates ................ 20
    26.4.5.1 Class template discard_block_engine .................. 21
    26.4.5.2 Class template independent_bits_engine ............... 22
    26.4.5.3 Class template shuffle_order_engine ................... 23
  26.4.6 Engines and engine adaptors with predefined parameters .... 24
  26.4.7 Class random_device ......................................... 26
  26.4.8 Utilities ..................................................... 27
    26.4.8.1 Class seed_seq ....................................... 27
    26.4.8.2 Function template generate_canonical .................. 29
  26.4.9 Random number distribution class templates .................. 30
    26.4.9.1 Uniform distributions .................................. 30
      26.4.9.1.1 Class template uniform_int_distribution ........... 30
      26.4.9.1.2 Class template uniform_real_distribution .......... 31
    26.4.9.2 Bernoulli distributions ................................ 32
      26.4.9.2.1 Class bernoulli_distribution ........................ 32
      26.4.9.2.2 Class template binomial_distribution ............... 33
      26.4.9.2.3 Class template geometric_distribution .............. 34
Concepts for Random Number Generation in C++0X (N2781)
Concepts for Random Number Generation in C++0X (N2781)
26  Numerics library

26.4  Random number generation

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

In addition to a few utilities, four categories of entities are described: uniform random number generators, random number engines, random number engine adaptors, and random number distributions. These categorizations are applicable to types that satisfy the corresponding 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 \(\text{bind}(d, e)\). — end note]

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 \(\text{numeric_limits}<T>::\text{is_integer}\) 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 \(T\).

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

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

a) the operator rshift denotes a bitwise right shift with zero-valued bits appearing in the high bits of the result, and

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

26.4.1  Requirements

Remove this section 26.4.1 [rand.req] in its entirety; its content has been recast in concept form and for the most part moved into the new section 26.4.3 [rand.concept].

26.4.2  Header <random> synopsis
namespace std {

    // 26.4.3.1 [rand.concept.urng] Concept UniformRandomNumberGenerator
    concept UniformRandomNumberGenerator<typename U> see below

    // 26.4.3.2 [rand.concept.eng] Concept RandomNumberEngine
    concept RandomNumberEngine<typename E> see below

    // 26.4.3.3 [rand.concept.adapt] Concept RandomNumberEngineAdaptor
    concept RandomNumberEngineAdaptor<typename A> see below

    // 26.4.3.4 [rand.concept.dist] Concept RandomNumberDistribution
    concept RandomNumberDistribution<typename D> see below

    // 26.4.3.5 [rand.concept.seedseq] Concept SeedSequence
    concept SeedSequence<typename S> see below

    // 26.4.4.1 [rand.eng.lcong] Class template linear_congruential_engine
    template<class UnsignedIntegralLike UIntType, UIntType a, UIntType c, UIntType m>
    requires IntegralType<UIntType>
    && True<m == 0u || (a < m && c < m)>
    class linear_congruential_engine;

    // 26.4.4.2 [rand.eng.mers] Class template mersenne_twister_engine
    template<class UnsignedIntegralLike UIntType, size_t w, size_t n, size_t m, size_t r,
            UIntType a, size_t u, size_t s,
            UIntType b, size_t t,
            UIntType c, size_t l>
    requires IntegralType<UIntType>
    && True<1u <= m && 1u <= n && 1u <= r && 1u <= u && 1u <= s && 1u <= t && 1u <= w &&
    w <= numeric_limits<UIntType>::digits &&
    a <= 2u<<w - 1u && b <= 2u<<w - 1u && c <= 2u<<w - 1u>
    class mersenne_twister_engine;

    // 26.4.4.3 [rand.eng.sub] Class template subtract_with_carry_engine
    template<class UnsignedIntegralLike UIntType, size_t w, size_t s, size_t r>
    requires IntegralType<UIntType>
    && True<s <= s < r && 0 < w && w <= numeric_limits<UIntType>::digits>
    class subtract_with_carry_engine;

    // 26.4.5.1 [rand.adapt/disc] Class template discard_block_engine
    template<class RandomNumberEngine Engine, size_t p, size_t r>
    requires True<1 <= r && r <= p>
    class discard_block_engine;

    // 26.4.5.2 [rand.adapt/ibits] Class template independent_bits_engine
    template<class RandomNumberEngine Engine, size_t w, class UnsignedIntegralLike UIntType>
    requires IntegralType<UIntType>
    && True<w <= numeric_limits<result_type>::digits>

    Concepts for Random Number Generation in C++0X (N2781)
class independent_bits_engine;

// 26.4.5.3 [rand.adapt.shuf] Class template shuffle_order_engine
template<class RandomNumberEngine Engine, size_t k>
  requires True<1u <= k>
class shuffle_order_engine;

// 26.4.6 [rand.predef] Engines and engine adaptors with predefined parameters
typedef see below minstd_rand0;
typedef see below minstd_rand;
typedef see below mt19937;
typedef see below ranlux24_base;
typedef see below ranlux48_base;
typedef see below ranlux24;
typedef see below ranlux48;
typedef see below knuth_b;
typedef see below default_random_engine;

// 26.4.7 [rand.device] Class random_device
class random_device;

// 26.4.8.1 [rand.util.seedseq] Class seed_seq
class seed_seq;

// 26.4.8.2 [rand.util.canonical] Function template generate_canonical
template<class FloatingPointLike RealType, size_t bits,
class UniformRandomNumberGenerator URNG>
  requires FloatingPointType<RealType>
  RealType generate_canonical(UniformRandomNumberGenerator URNG & g);

// 26.4.9.1.1 [rand.dist.uni.int] Class template uniform_int_distribution
template<class IntegralLike IntType = int>
  requires IntegralType<IntType>
class uniform_int_distribution;

// 26.4.9.1.2 [rand.dist.uni.real] Class template uniform_real_distribution
template<class FloatingPointLike RealType = double>
  requires FloatingPointType<RealType>
class uniform_real_distribution;

// 26.4.9.2.1 [rand.dist.bern.bernoulli] Class bernoulli_distribution
class bernoulli_distribution;

// 26.4.9.2.2 [rand.dist.bern.bin] Class template binomial_distribution
template<class IntegralLike IntType = int>
  requires IntegralType<IntType>
class binomial_distribution;

// 26.4.9.2.3 [rand.dist.bern.geo] Class template geometric_distribution
template<class IntegralLike IntType = int>
  requires IntegralType<IntType>

Concepts for Random Number Generation in C++0X (N2781)
class geometric_distribution;

// 26.4.9.2.4 [rand.dist.bern.negbin] Class template negative_binomial_distribution
template<class IntegralLike IntType = int>
    requires IntegralType<IntType>
    class negative_binomial_distribution;

// 26.4.9.3.1 [rand.dist.pois.poisson] Class template poisson_distribution
template<class IntegralLike IntType = int>
    requires IntegralType<IntType>
    class poisson_distribution;

// 26.4.9.3.2 [rand.dist.pois.exp] Class template exponential_distribution
template<class FloatingPointLike RealType = double>
    requires FloatingPointType<RealType>
    class exponential_distribution;

// 26.4.9.3.3 [rand.dist.pois.gamma] Class template gamma_distribution
template<class FloatingPointLike RealType = double>
    requires FloatingPointType<RealType>
    class gamma_distribution;

// 26.4.9.3.4 [rand.dist.pois.weibull] Class template weibull_distribution
template<class FloatingPointLike RealType = double>
    requires FloatingPointType<RealType>
    class weibull_distribution;

// 26.4.9.3.5 [rand.dist.pois.extreme] Class template extreme_value_distribution
template<class FloatingPointLike RealType = double>
    requires FloatingPointType<RealType>
    class extreme_value_distribution;

// 26.4.9.4.1 [rand.dist.norm.normal] Class template normal_distribution
template<class FloatingPointLike RealType = double>
    requires FloatingPointType<RealType>
    class normal_distribution;

// 26.4.9.4.2 [rand.dist.norm.lognormal] Class template lognormal_distribution
template<class FloatingPointLike RealType = double>
    requires FloatingPointType<RealType>
    class lognormal_distribution;

// 26.4.9.4.3 [rand.dist.norm.chisq] Class template chi_squared_distribution
template<class FloatingPointLike RealType = double>
    requires FloatingPointType<RealType>
    class chi_squared_distribution;

// 26.4.9.4.4 [rand.dist.norm.cauchy] Class template cauchy_distribution
template<class FloatingPointLike RealType = double>
    requires FloatingPointType<RealType>

Concepts for Random Number Generation in C++0X (N2781)
class cauchy_distribution;

// 26.4.9.4.5 [rand.dist.norm.f] Class template fisher_f_distribution
template<
class FloatingPointLike RealType = double>
    requires FloatingPointType<RealType>
class fisher_f_distribution;

// 26.4.9.4.6 [rand.dist.norm.t] Class template student_t_distribution
template<
class FloatingPointLike RealType = double>
    requires FloatingPointType<RealType>
class student_t_distribution;

// 26.4.9.5.1 [rand.dist.samp.discrete] Class template discrete_distribution
template<
class IntegralLike IntType = int>
    requires IntegralType<IntType>
class discrete_distribution;

// 26.4.9.5.2 [rand.dist.samp.pconst] Class template piecewise_constant_distribution
template<
class FloatingPointLike RealType = double>
    requires FloatingPointType<RealType>
class piecewise_constant_distribution;

// 26.4.9.5.3 [rand.dist.samp.genpdf] Class template general_pdf_distribution
template<
class FloatingPointLike RealType = double>
    requires FloatingPointType<RealType>
class general_pdf_distribution;
} // namespace std

26.4.3 Concepts and related requirements for random number generation

26.4.3.1 Concept UniformRandomNumberGenerator

A uniform random number generator \( g \) of type \( G \) is a function object returning unsigned integral values such that each value in the range of possible results has (ideally) equal probability of being returned. [Note: The degree to which \( g \)'s results approximate the ideal is often determined statistically. —end note]

The following concept may be overconstrained, as its IntegralType<result_type> requirement does not allow a URNG that returns a value of user-defined unsigned integral type \( T \). We should instead merely require that \( T \) have the expected unsigned integer behaviors (arithmetic, interoperability, etc.). However, additional groundwork seems needed before we can do so. (Papers N2645 = Fundamental Mathematical Concepts ... and N2650 = Toward a More Complete Taxonomy ... explicate some of the necessary concepts.)

concept UniformRandomNumberGenerator<
typename G> : Callable<G> {
    requires UnsignedIntegralLike<result_type>
    \&\& IntegralType<result_type>;

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

    Concepts for Random Number Generation in C++0X (N2781)
26.4 Random number generation

```
axiom NonemptyRange(G& g) {
    min() < max();
}
axiom InRange(G& g) {
    min() <= g() && g() <= max();
}

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

2 Complexity: amortized constant.
```

26.4.3.2 Concept RandomNumberEngine

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

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

The input/output requirements of this and subsequent sections are specified in terms of stream concepts InputStreamable and OutputStreamable that are planned for C++0X, but that have not to date been formally proposed.

```
concept RandomNumberEngine<typename E> : Regular<E>, UniformRandomNumberGenerator<E> {
    requires Constructible<E, result_type>;
    template<SeedSequence Sseq> E::E(Sseq& q);

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

    void discard(E& e, unsigned long long z) {
        for( ; z > 0ULL; --z) e();
    }

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

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

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

Concepts for Random Number Generation in C++0X (N2781)
At any given time, \( e \) has a state \( e_i \) for some integer \( i \geq 0 \). Upon construction, \( e \) has an initial state \( e_0 \). An engine’s state may be established via a constructor, a \texttt{seed} member function, assignment, or a suitable \texttt{operator}>>.

E’s specification shall define:

a) the size of E’s state in multiples of the size of \texttt{result\_type}, given as an integral constant expression;

b) the \textit{transition algorithm} \( \text{TA} \) by which \( e \)’s state \( e_i \) is advanced to its \textit{successor state} \( e_{i+1} \); and

c) the \textit{generation algorithm} \( \text{GA} \) by which an engine’s state is mapped to a value of type \texttt{result\_type}.

\begin{verbatim}
bool operator==(const E& e1, const E& e2); // from Regular<E>
\end{verbatim}

Returns: true if \( S_1 = S_2 \), where \( S_1 \) and \( S_2 \) are the infinite sequences of values that would be generated, respectively, by repeated future calls to \( e1() \) and \( e2() \). Otherwise returns \texttt{false}.

\begin{verbatim}
void operator() (E& e); // from UniformRandomNumberGenerator<E>
\end{verbatim}

Effects: Sets the state to \( e_{i+1} = \text{TA}(e_i) \).

Returns: \( \text{GA}(e_i) \).

\begin{verbatim}
E::E(result_type s); // from Constructible<E,result_type>
\end{verbatim}

Effects: Creates an engine with an initial state that depends on \( s \).

\begin{verbatim}
template<SeedSequence Sseq> E::E(Sseq& q); 
\end{verbatim}

Effects: Creates an engine with an initial state that depends on a sequence produced by one call to \( q.generate \).

\begin{verbatim}
void seed(E& e); 
\end{verbatim}

Complexity: same as \( \text{E}() \).

\begin{verbatim}
void seed(E& e, result_type s); 
\end{verbatim}

Complexity: same as \( \text{E}(s) \).

\begin{verbatim}
template<SeedSequence Sseq> void seed(E& e, Sseq& q); 
\end{verbatim}

Complexity: same as \( \text{E}(q) \).

\begin{verbatim}
void discard(E& e, unsigned long long z); 
\end{verbatim}

Effects: Advances the engine’s state from \( e_i \) to \( e_{i+z} \) by any means equivalent to the default implementation specified above.

Concepts for Random Number Generation in C++0X (N2781)
26.4 Random number generation

Complexity: no worse than the complexity of \( z \) consecutive calls to \( \text{operator}()() \).

Note: This operation is common in user code, and can often be implemented in an engine-specific manner so as to provide significant performance improvements over the default implementation specified above.

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

Effects: With \( \text{os}.\text{fmtflags} \) set to \( \text{ios\_base::dec} | \text{ios\_base::left} \) and the fill character set to the space character, writes to \( \text{os} \) the textual representation of \( e \)'s current state. In the output, adjacent numbers are separated by one or more space characters.

Returns: the updated \( \text{os} \).

Postcondition: The \( \text{os}.\text{fmtflags} \) and fill character are unchanged.

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

Requires: \( \text{is} \) provides a textual representation that was previously written using an output stream whose imbued locale was the same as that of \( \text{is} \), and whose associated types \( \text{OutputStreamable::charT} \) and \( \text{OutputStreamable::traits} \) were respectively the same as those of \( \text{is} \).

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

Returns: the updated \( \text{is} \).

Postcondition: The \( \text{is}.\text{fmtflags} \) are unchanged.

26.4.3.3 Concept RandomNumberEngineAdaptor [rand.concept.adapt]

A random number engine adaptor (commonly shortened to adaptor) \( a \) of type \( A \) is a random number engine that takes values produced by some other random number engine or engines, and applies an algorithm to those values in order to deliver a sequence of values with different randomness properties. Engines 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.

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

```cpp
class RandomNumberEngineAdaptor<
    typename A>
{
    requires Constructible<A, const RandomNumberEngine&...>
    && Constructible<A, RandomNumberEngine&&...>;
}
```

Concepts for Random Number Generation in C++0X (N2781)
bool operator==(const A& a1, const A& a2); // from RandomNumberEngine<A>

Returns: true if each pair of corresponding b_i are equal. Otherwise returns false.

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

Effects: Each b_i is initialized, in turn, with the next available value from the list s + 0, s + 1, ... .

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

Effects: Each b_i is initialized, in turn, with q as argument.

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

Effects: For each b_i, in turn, invokes b_i.seed() .

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

Effects: For each b_i, in turn, invokes b_i.seed(s) with the next available value from the list s + 0, s + 1, ... .

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

Effects: For each b_i, in turn, invokes b_i.seed(q) .

A shall also satisfy the following additional requirements:

a) The complexity of each function shall be at most the sum of the complexities of the corresponding functions applied to each base engine.

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

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

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

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

In addition to the adaptor requirements articulated above, the Working Draft has one more: that each adaptor shall contain typedefs that enumerate the types of each of the adaptor’s base engines. The omission from this paper is deliberate.

The omitted requirement was originally intended to make it possible to write generic algorithms over engine adaptors, following the general principle that a template should publish all its template parameters. However, noting that the standard adaptors are all named *_engine, it became clear that the salient feature of an adaptor is that it behaves as an engine. That it adapts other engines in order to do so is incidental to an adaptor’s generic use.
26.4 Random number generation

Other than introspection for its own sake, we have found no use for the omitted typedefs: we have found no algorithms restricted to engines that happen to be adaptors. This paper therefore proposes that the omitted base engine typedefs be permanently deleted from engine adaptor requirements.

26.4.3.4 Concept RandomNumberDistribution

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

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

Some view this concept’s result_type as overconstrained because it demands an ArithmeticType and thus, for example, disallows returning a container of results. However, the specified behavior is of long standing (see 26.4 [random.numbers] paragraph 3) and so should not be relaxed without additional careful consideration.

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

    void reset(D& d);

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

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

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

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

Regular param_type;

Requires:

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

Concepts for Random Number Generation in C++0X (N2781)
b) For each of the member functions of $D$ that return values corresponding to parameters of the distribution,\n\n$param\_type$ shall provide a corresponding member function with the identical name, type, and semantics.

c) $param\_type$ shall provide a declaration of the form typedef $D$ distribution_type;

Remark: It is unspecified whether $param\_type$ is declared as a (nested) class or via a typedef. In this subclause 26.4 [random.numbers], declarations of $D::param\_type$ are in the form of typedefs only for convenience of exposition.

```cpp
D::D(const param_type& p);
```

Effects: Creates a distribution whose behavior is indistinguishable from that of a distribution newly created directly from the values used to create $p$.

Complexity: same as $p$'s construction.

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

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

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

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

Complexity: constant.

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

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

For distributions $x$ and $y$ of identical type $D$:

a) The sequence of numbers produced by repeated invocations of $x(u)$ shall be independent of any invocation of $os << x$ or of any const member function of $D$ between any of the invocations $x(u)$.

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

Complexity: amortized constant number of invocations of $u$.

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

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

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

Returns: a value $p$ such that $param(D(p)) == p$.

Complexity: no worse than the complexity of $D(p)$. 

Concepts for Random Number Generation in C++0X (N2781)
void param(D& d, const param_type& p);

Postcondition: \( \text{param}(D(), p) == p \).

Complexity: no worse than the complexity of \( D(p) \).

result_type min(const D& d);

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

Complexity: constant.

result_type max(const D& d);

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

Complexity: constant.

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

Effects: Writes to \( os \) a textual representation for the parameters and the additional internal data of \( d \).

Returns: the updated \( os \).

Postcondition: The \( os \).fmtflags and fill character are unchanged.

template<InputStreamable IS> IS& operator>>(IS& is, D& d);

Requires: \( is \) provides a textual representation that was previously written using an output stream whose imbued locale was the same as that of \( is \), and whose associated types OutputStreamable::charT and OutputStreamable::traits were respectively the same as those of \( is \).

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

Returns: the updated \( is \).

Postcondition: The \( is \).fmtflags are unchanged.

26.4.3.5 Concept SeedSequence

This section 26.4.3.5 [rand.concept.seedseq] has no precise analog in the Working Draft. Rather, the section abstracts the 26.4.8.1 [rand.util.seedseq] requirements originally imposed on the seed_seq class. By creating this separate concept (and adjusting the RandomNumberEngine concept accordingly), we give users latitude to provide their own seed sequence types, rather than forcing all users to rely solely on the single seed_seq specified in the Working Draft.
Although not rising to the level of a defect, a number of users have voiced concerns regarding the Working Paper’s lack of such flexibility. These concerns have to date been addressed (for example, in LWG issue 731) via a gentlemen’s agreement that the desired customization point would be provided via a concept facility once one became viable. We now provide this subsection to fulfill that promise in a manner consistent with users’ long-standing latitude to provide their own interoperable generators, engines, adaptors, and distributions.

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

```cpp
concept SeedSequence<typename S> : Semiregular<S>, DefaultConstructible<S> {
    UnsignedIntegralLike result_type;
    requires IntegralType<result_type>
    && True<sizeof uint32_t <= sizeof result_type>;

    template<InputIterator Iter>
    requires IntegralLike<Iter::value_type>
    && IntegralType<Iter::value_type>
    S::S(Iter begin, Iter end, size_t u = numeric_limits<Iter::value_type>::digits);

    template<RandomAccessIterator Iter>
    requires UnsignedIntegralLike<Iter::value_type>
    && IntegralType<Iter::value_type>
    && True<sizeof uint32_t <= sizeof Iter::value_type>
    void generate(S& q, Iter begin, Iter end);

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

2 Effects: Constructs a SeedSequence object having internal state that depends on some or all of the bits of the supplied sequence [begin, end).

```cpp
template<InputIterator Iter>
requires IntegralLike<Iter::value_type>
&& IntegralType<Iter::value_type>
S::S(Iter begin, Iter end,
    size_t u = numeric_limits<Iter::value_type>::digits);
```

3 Effects: Does nothing if begin == end. Otherwise, fills the supplied range [begin,end) with 32-bit quantities that depend on the sequence supplied to the constructor and possibly also on the history of generate’s previous

Concepts for Random Number Generation in C++0X (N2781)
26.4 Random number generation Numerics library

size_t size(const S& q);

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

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

Effects: Copies to the given destination a sequence of 32-bit units that can be provided to the constructor of a second object of the same type, and that would reproduce in that second object a state indistinguishable from the state of the first object.

26.4.4 Random number engine class templates

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

Except as required by Table clause ref tab RandomEngine, where specified otherwise, no function described in this section 26.4.4 [rand.eng] throws an exception.

The class templates specified in this section 26.4.4 [rand.eng] satisfy the requirements of random number engine ?? [rand.req.eng] For every class E instantiated from a template specified in this section 26.4.4 [rand.eng], a concept map RandomNumberEngine<E> shall be defined in namespace std so as to provide mappings from free functions to the corresponding member functions. Descriptions are provided here only for engine operations on the engines that are not described in 26.4.3.2 [rand.concept.eng] 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 linear_congruential_engine

A linear_congruential_engine random number engine produces unsigned integer random numbers. The state \( x_i \) of a linear_congruential_engine object \( x \) is of size 1 and consists of a single integer. The transition algorithm is a modular linear function of the form \( TA(x_i) = (a \cdot x_i + c) \mod m \); the generation algorithm is \( GA(x_i) = x_{i+1} \).

template<class UnsignedIntegralLike UIntType, UIntType a, UIntType c, UIntType m>
requires IntegralType<UIntType>
&& True<m == 0u \| (a < m && c < m)>
class linear_congruential_engine
{
public:

// types
typedef UIntType result_type;

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

Concepts for Random Number Generation in C++0X (N2781)
26.4 Random number generation

```cpp
// constructors and seeding functions
explicit linear_congruential_engine(result_type s = default_seed);
template<SeedSequence Sseq> explicit linear_congruential_engine<double , result_type s = default_seed);
void seed(result_type s = default_seed);
template<SeedSequence Sseq> void seed(double , seed_sequence Sseq& q);

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

The textual representation consists of the value of $x_i$.

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

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

```cpp
template<SeedSequence Sseq> explicit linear_congruential_engine<double , result_type s = default_seed>
```

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

26.4.4.2 Class template `merсенne_twister_engine`

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

The transition algorithm employs a twisted generalized feedback shift register defined by shift values $n$ 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 $u, s, b, t, c, \text{ and } \ell$.

The state transition is performed as follows:

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

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

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

---

1) 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.
b) Let \( z_2 = z_1 \text{xor} \left( (z_1 \text{lshift}_w s) \text{bitand} b \right) \).

c) Let \( z_3 = z_2 \text{xor} \left( (z_2 \text{lshift}_w t) \text{bitand} c \right) \).

d) Let \( z_4 = z_3 \text{xor} \left( z_3 \text{rshift}_\ell \right) \).

template<class UnsignedIntegralLike UIntType, size_t w, size_t n, size_t m, size_t r,
UIntType a, size_t u, size_t s,
UIntType b, size_t t,
UIntType c, size_t l>
requires IntegralType<UIntType>
&& True<1u <= m && 1u <= n
&& r <= w && u <= w && s <= w && t <= w && l <= w && w <= numeric_limits<UIntType>::digits
&& a <= 2u<<w - 1u && b <= 2u<<w - 1u && c <= 2u<<w - 1u>

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

// engine characteristics
static const size_t word_size = w;
static const size_t state_size = n;
static const size_t shift_size = m;
static const size_t mask_bits = r;
static const UIntType xor_mask = a;
static const size_t tempering_u = u;
static const size_t tempering_s = s;
static const UIntType tempering_b = b;
static const size_t tempering_t = t;
static const UIntType tempering_c = c;
static const size_t tempering_l = l;
static constexpr result_type min () { return 0; }
static constexpr result_type max() { return 2w - 1;
static const result_type default_seed = 5489u;

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

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

The following relations shall hold: \( 1 \leq m \leq n; 0 \leq r, u, s, t, l \leq \text{numeric\_limits<result\_type>::digits}; 0 \leq a, b, c \leq 2^w - 1 \).

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

Concepts for Random Number Generation in C++0X (N2781)
explicit mersenne_twister_engine(result_type value = default_seed);

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

\[
[1812433253 \cdot (X_{i-1} \text{xor} (X_{i-1} \text{rshift}(w-2))) + i \mod n] \mod 2^w
\]

Complexity: \(O(n)\).

template<SeedSequence Sseq> explicit mersenne_twister_engine(seed_seq Sseq & q);

Effects: Constructs a mersenne_twister_engine object. With \(k = \lceil w/32 \rceil\) and \(a\) an array (or equivalent) of length \(n \cdot k\), invokes \(q.generate(a + 0, a + n \cdot k)\) and then, iteratively for \(i = -n, \ldots, -1\), sets \(X_i\) to

\[
\left(\sum_{j=0}^{k-1} a_{k(i+n)+j} \cdot 2^{32j}\right) \mod 2^w.
\]

Finally, if the most significant \(w - r\) bits of \(X_{-n}\) are zero, and if each of the other resulting \(X_i\) is 0, changes \(X_{-n}\) to \(2^w - 1\).

26.4.4.3 Class template subtract_with_carry_engine

A subtract_with_carry_engine random number engine produces unsigned integer random numbers.

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

The state transition is performed as follows:

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

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

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

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

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

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

  Concepts for Random Number Generation in C++0X (N2781)
static const result_type default_seed = 19780503u;

// constructors and seeding functions
explicit subtract_with_carry_engine(result_type value = default_seed);

template<SeedSequence Sseq> explicit subtract_with_carry_engine(seed_seq& q);

void seed(result_type value = default_seed);

template<SeedSequence Sseq> void seed(seed_seq& q);

// generating functions
result_type operator()();

void discard(unsigned long long z);

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

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

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

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

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

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

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

Effects: Constructs a subtract_with_carry_engine object. With $k = \lfloor w/32 \rfloor$ and $a$ an array (or equivalent) of length $r \cdot k$, invokes $q.generate(a + 0, a + r \cdot k)$ and then, iteratively for $i = -r, \ldots, -1$, sets $X_i$ to $\left(\sum_{j=0}^{k-1} a_{k(i+r)+j} \cdot 2^{32j}\right) \mod m$. If $X_{i-1}$ is then 0, sets $c$ to 1; otherwise sets $c$ to 0.
Declarations for copy constructors, for copy assignment operators, and for equality and inequality operators are not shown in the synopses.

26.4.5.1 Class template discard_block_engine  

A discard_block_engine random number engine adaptor produces random numbers selected from those produced by some base engine \( e \). The state \( x_i \) of a discard_block_engine engine adaptor object \( x \) consists of the state \( e_i \) of its base engine \( e \) and an additional integer \( n \). The size of the state is the size of \( e \)’s state plus 1.

The transition algorithm discards all but \( r > 0 \) values from each block of \( p \geq r \) values delivered by \( e \). The state transition is performed as follows: If \( n \geq r \), advance the state of \( e \) from \( e_i \) to \( e_{i+p-r} \) and set \( n \) to 0. In any case, then increment \( n \) and advance \( e \)’s then-current state \( e_j \) to \( e_{j+1} \).

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

```cpp
template<class RandomNumberEngine Engine, size_t p, size_t r>
requires True<1 <= r && r <= p>
class discard_block_engine
{
public:
    // types
    typedef Engine base_type;
    typedef typename base_type::result_type result_type;

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

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

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

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

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

Concepts for Random Number Generation in C++0X (N2781)
The following relations shall hold: \( 1 \leq r \leq p \).

The textual representation consists of the textual representation of \( e \) followed by the value of \( n \).

In addition to its behavior pursuant to section 26.4.3.3 [rand.concept.adapt], each constructor that is not a copy constructor sets \( n \) to 0.

26.4.5.2 Class template independent_bits_engine [rand.adapt.ibits]

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

The transition and generation algorithms are described in terms of the following integral constants:

a) Let \( R = e.\text{max}() - e.\text{min}() + 1 \) and \( m = \lceil \log_2 R \rceil \).

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

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

[Note: The relation \( w = n_0w_0 + (n - n_0)(w_0 + 1) \) always holds. — end note]

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

The generation algorithm uses the values produced while advancing the state as described above to yield a quantity \( S \) obtained as if by the following algorithm:

\[
S = 0; \\
\text{for } (k = 0; k \neq n_0; k += 1) \{ \\
\quad \text{do } u = e() - e.\text{min}(); \text{ while } (u \geq y_0); \\
\quad S = 2^{w_0} \cdot S + u \mod 2^{w_0}; \\
\} \\
\text{for } (k = n_0; k \neq n; k += 1) \{ \\
\quad \text{do } u = e() - e.\text{min}(); \text{ while } (u \geq y_1); \\
\quad S = 2^{w_0+1} \cdot S + u \mod 2^{w_0+1}; \\
\}
\]

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

class independent_bits_engine
{
public:

  // types
  typedef Engine base_type;
  typedef UIntType result_type;

  // engine characteristics

  Concepts for Random Number Generation in C++0X (N2781)
static constexpr result_type min() { return 0; }
static constexpr result_type max() { return \(2^n - 1\); }

// constructors and seeding functions
independent_bits_engine();
explicit independent_bits_engine(const \texttt{base\_type Engine}\& e);
explicit independent_bits_engine(Engine\&\& e);
explicit independent_bits_engine(result_type s);
template<SeedSequence Sseq> explicit independent_bits_engine(\texttt{seed\_seq}\& q);
void seed();
void seed(result_type s);
template<SeedSequence Sseq> void seed(\texttt{seed\_seq}\& q);

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

// property functions
const \texttt{base\_type Engine}\& base() const;

private:
    \texttt{base\_type Engine} e; // exposition only
};

The following relations shall hold: \(0 < w \leq \text{numeric\_limits<result\_type>::digits}\).

The textual representation consists of the textual representation of \texttt{e}.

26.4.5.3 Class template \texttt{shuffle\_order\_engine} [rand.adapt.shuf]

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

The transition algorithm permutes the values produced by \(e\). The state transition is performed as follows:

a) Calculate an integer \(j\) as \(\left\lfloor \frac{k(Y - \text{min})}{\text{max} - \text{min}} + 1 \right\rfloor\).

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

The generation algorithm yields the last value of \(Y\) produced while advancing \(e\)'s state as described above.
typedef typename base_typeEngine::result_type result_type;

// engine characteristics
static const size_t table_size = k;
static constexpr result_type min() { return base_typeEngine::min; }
static constexpr result_type max() { return base_typeEngine::max; }

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

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

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

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

The following relation shall hold: 1 \leq k.

The textual representation consists of the textual representation of e, followed by the k values of V, followed by the value of Y.

In addition to its behavior pursuant to section 26.4.3.3 [rand.concept.adapt], each constructor that is not a copy constructor initializes V[0], ..., V[k-1] and Y, in that order, with values returned by successive invocations of e().

26.4.6 Engines and engine adaptors with predefined parameters

typedef linear_congruential_engine<uint_fast32_t, 16807, 0, 2147483647> minstd_rand0;

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

typedef linear_congruential_engine<uint_fast32_t, 48271, 0, 2147483647> minstd_rand;

Concepts for Random Number Generation in C++0X (N2781)
### 26.4 Random number generation

#### Required behavior

2. The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type `minstd_rand` shall produce the value 399268537.

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

3. The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type `mt19937` shall produce the value 4123659995.

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

4. The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type `ranlux24_base` shall produce the value 7937952.

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

5. The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type `ranlux48_base` shall produce the value 618391285275.

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

6. The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type `ranlux24` shall produce the value 9901578.

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

7. The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type `ranlux48` shall produce the value 249142670248501.

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

8. The 10000\textsuperscript{th} consecutive invocation of a default-constructed object of type `knuth_b` shall produce the value 1112339016.

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

#### Remark

The choice of engine type named by this typedef is implementation defined. [Note: The implementation may select this type on the basis of performance, size, quality, or any combination of such factors, so as to provide at least acceptable engine behavior for relatively casual, inexpert, and/or lightweight use. Because different

### Concepts for Random Number Generation in C++0X (N2781)
implementations may select different underlying engine types, code that uses this typedef need not generate identical sequences across implementations. — end note]

26.4.7 Class random_device

A random_device uniform random number generator produces non-deterministic random numbers. It satisfies the requirements of uniform random number generator [rand.req.urng] a concept map UniformRandomNumberGenerator<random_device> shall be defined in namespace std so as to provide mappings from free functions to the corresponding member functions.

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

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

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

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

        // generating functions
        result_type operator()();

        // property functions
        double entropy() const;

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

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.

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

Effects: Constructs a random_device non-deterministic uniform random number generator object. The semantics and default value of the token parameter are implementation-defined.  

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

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

2) The parameter is intended to allow an implementation to differentiate between different sources of randomness.

Concepts for Random Number Generation in C++0X (N2781)
Returns: If the implementation employs a random number engine, returns 0.0. Otherwise, returns an entropy estimate for the random numbers returned by `operator()`, in the range `min()` to \( \log_2(\max() + 1) \).

Throws: Nothing.

result_type operator()();

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

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

26.4.8 Utilities

26.4.8.1 Class seed_seq

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

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

No function described in this section 26.4.8.1 [rand.util.seedseq] throws an exception.

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

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

    // constructors
    seed_seq();
    template<
        class InputIterator Iter>
        requires IntegralLike<Iter::value_type>
        && IntegralType<Iter::value_type>
        seed_seq(InputIterator begin, InputIterator end,
        size_t u = numeric_limits<typename iterator_traits<Iter>::value_type>::digits);

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

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

Concepts for Random Number Generation in C++0X (N2781)
// property functions
size_t size() const;
template<class OutputIterator<auto,const result_type&> Iter>
void param(OutputIterator<Iter> dest) const;

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

explicit seed_seq();

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

Throws: Nothing.

template<class InputIterator Iter>
requires IntegralLike<Iter::value_type>
& IntegralType<Iter::value_type>
seed_seq(InputIterator begin, InputIterator end,
    size_t u = numeric_limits<typename iterator_traits<Iter>::value_type>::digits);

Requires: InputIterator shall satisfy the requirements of an input iterator [input.iterators] such that iterator_traits<Iter>::value_type shall denote an integral type.

Effects: Constructs a seed_seq object by rearranging some or all of the bits of the supplied sequence [begin, end) of w-bit quantities into 32-bit units, as if by the following:

First extract the rightmost u bits from each of the \( n = \text{end} - \text{begin} \) elements of the supplied sequence and concatenate all the extracted bits to initialize a single (possibly very large) unsigned binary number, \( b = \sum_{i=0}^{n-1} (\text{begin}[i] \mod 2^u) \cdot 2^{w \cdot i} \) (in which the bits of each \( \text{begin}[i] \) are treated as denoting an unsigned quantity). Then carry out the following algorithm:

- \( v.\text{clear}() \);
- if \( (w < 32) \)
  - \( v.\text{push\_back}(n) \);
- for( ; n > 0; --n )
  - \( v.\text{push\_back}(b \mod 2^{32}), b /= 2^{32}; \)


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

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

Effects: Does nothing if begin == end. Otherwise, with \( s = v.\text{size()} \) and \( n = \text{end} - \text{begin} \), fills the supplied range [begin, end) according to the following algorithm in which each operation is to be carried out modulo \( 2^{32} \), each indexing operator applied to begin is to be taken modulo \( n \), and \( T(x) \) is defined as \( x \text{ xor} (x \text{ rshift} 27) \):
29 Numerics library 26.4 Random number generation

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

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

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

\[
\begin{align*}
r_1 &= 1664525 \cdot T(\text{begin}[k] \boxtimes \text{begin}[k+p] \boxtimes \text{begin}[k-1]) \\
r_2 &= r_1 + \begin{cases} 
s, & 0 \leq k \leq s \\
k \mod n + v[k-1], & s < k
\end{cases}
\end{align*}
\]

and, in order, increment \( \text{begin}[k+p] \) by \( r_1 \), increment \( \text{begin}[x+q] \) by \( r_2 \), and set \( \text{begin}[k] \) to \( r_2 \).

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

\[
\begin{align*}
r_3 &= 1566083941 \cdot T(\text{begin}[k] + \text{begin}[k+p] + \text{begin}[k-1]) \\
r_4 &= r_3 - (k \mod n)
\end{align*}
\]

and, in order, update \( \text{begin}[k+p] \) by xoring it with \( r_4 \), update \( \text{begin}[x+q] \) by xoring it with \( r_3 \), and set \( \text{begin}[k] \) to \( r_4 \).

**Throws:** Nothing.

size_t size() const;

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

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

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

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

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

26.4.8.2 Function template generate_canonical [rand.util.canonical]

Each function instantiated from the template described in this section 26.4.8.2 [rand.util.canonical] maps the result of one or more invocations of a supplied uniform random number generator \( g \) to one member of the specified RealType such that, if the values \( g_i \) produced by \( g \) are uniformly distributed, the instantiation’s results \( t_j, 0 \leq t_j < 1 \), are distributed as uniformly as possible as specified below.

[Note: Obtaining a value in this way can be a useful step in the process of transforming a value generated by a uniform random number generator into a value that can be delivered by a random number distribution. — end note]
template<
    class FloatingPointLike RealType,
    size_t bits,
    class UniformRandomNumberGenerator URNG>
requires FloatingPointType<RealType>
RealType generate_canonical(UniformRandomNumberGenerator URNG& g);

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

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

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

using arithmetic of type RealType.

5 Returns: $S/R^k$.

6 Throws: What and when $g$ throws.

### 26.4.9 Random number distribution class templates [rand.dist]

The classes and class templates specified in this section 26.4.9 satisfy all the requirements of random number distribution [rand.concept.dist] For every class $D$ specified in this section 26.4.9 [rand.dist] or instantiated from a template specified in this section, a concept map RandomNumberDistribution<$D$> shall be defined in namespace std so as to provide mappings from free functions to the corresponding member functions. Descriptions are provided here only for distribution operations on the distributions that are not described in 26.4.3.4 [rand.concept.dist] 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.

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

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.

#### 26.4.9.1 Uniform distributions [rand.dist.uni]

#### 26.4.9.1.1 Class template uniform_int_distribution [rand.dist.uni.int]

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

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

4) $b$ is introduced to avoid any attempt to produce more bits of randomness than can be held in RealType.
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 URNG>
result_type operator()(UniformRandomNumberGenerator URNG& g);
template<class UniformRandomNumberGenerator URNG>
result_type operator()(UniformRandomNumberGenerator URNG& g, const param_type& parm);

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

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

2 Requires: a \leq b.

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

result_type a() const;

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

result_type b() const;

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

26.4.9.1.2 Class template uniform_real_distribution

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

\[ p(x|a,b) = \frac{1}{b-a}. \]

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

        Concepts for Random Number Generation in C++0X (N2781)
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 URNG>
result_type operator()(UniformRandomNumberGenerator URNG& g);
template<class UniformRandomNumberGenerator URNG>
result_type operator()(UniformRandomNumberGenerator URNG& g, const param_type& parm);

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

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

2 Requires: \(a \leq b\) and \(b - a \leq \text{numeric\_limits<}\text{RealType}>\cdot\text{max()}\).

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

result_type a() const;

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

result_type b() const;

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

26.4.9.2 Bernoulli distributions [rand.dist.bern]

26.4.9.2.1 Class bernoulli_distribution [rand.dist.bern.bernoulli]

A bernoulli\_distribution random number distribution produces bool values \(b\) distributed according to the discrete probability function

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

class bernoulli_distribution
{
  public:

  Concepts for Random Number Generation in C++0X (N2781)
// 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 URNG>
result_type operator() (UniformRandomNumberGenerator URNG& g);

// 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);

Requires: 0 \leq p \leq 1.

Effects: Constructs a bernoulli_distribution object; p corresponds to the parameter of the distribution.

double p() const;

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

26.4.9.2.2 Class template binomial_distribution

A binomial_distribution random number distribution produces integer values \( i \geq 0 \) distributed according to the discrete probability function

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

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

    // constructors and reset functions

    Concepts for Random Number Generation in C++0X (N2781)
26.4 Random number generation

explicit binomial_distribution(IntType t = 1, double p = 0.5);
explicit binomial_distribution(const param_type& parm);
void reset();

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

// property functions
IntType t() const;
  Returns: The value of the t parameter with which the object was constructed.
double p() const;
  Returns: The value of the p parameter with which the object was constructed.

26.4.9.2.3 Class template geometric_distribution

A geometric_distribution random number distribution produces integer values \( i \geq 0 \) distributed according to the discrete probability function
\[
P(i|p) = p \cdot (1 - p)^i.
\]

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

    // constructors and reset functions

  Concepts for Random Number Generation in C++0X (N2781)
explicit geometric_distribution(double p = 0.5);
explicit geometric_distribution(const param_type& parm);
void reset();

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

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

explicit geometric_distribution(double p = 0.5);

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

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

26.4.9.2.4 Class template negative_binomial_distribution

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

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

// generating functions
#define RANDLINT k
#define RANDDOUBLE p
#define RANDPARAMETERSET param

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

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

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

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

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

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

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

26.4.9.3 Poisson distributions

26.4.9.3.1 Class template poisson_distribution

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

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

The distribution parameter \( \mu \) is also known as this distribution’s mean.

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

Concepts for Random Number Generation in C++0X (N2781)
// 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 URNG>
result_type operator()(
    UniformRandomNumberGenerator URNG& g);

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

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

explicit poisson_distribution(double mean = 1.0);

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

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

26.4.9.3.2 Class template exponential_distribution [rand.dist.pois.exp]

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

\[
p(x|\lambda) = \lambda e^{-\lambda x}.
\]

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

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

    // generating functions

Concepts for Random Number Generation in C++0X (N2781)
26.4 Random number generation

Numerics library

template<class UniformRandomNumberGenerator URNG>
result_type operator()(UniformRandomNumberGeneratorURNG& g);

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

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

explicit exponential_distribution(RealType lambda = 1.0);

Requires: 0 < lambda.

Effects: Constructs an exponential_distribution object; lambda corresponds to the parameter of the distribution.

RealType lambda() const;

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

26.4.9.3.3 Class template gamma_distribution

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

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

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

// types
typedef RealType result_type;
typedef unspecified param_type;

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

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

Concepts for Random Number Generation in C++0X (N2781)
result_type operator()(UniformRandomNumberGeneratorURNG& g, const param_type& parm);

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

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

Requires: 0 < alpha and 0 < beta.

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

RealType alpha() const;

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

RealType beta() const;

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

26.4.9.3.4 Class template weibull_distribution

A weibull_distribution random number distribution produces random numbers x ≥ 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 FloatingPointLike RealType = double>
requires FloatingPointType<RealType>
class weibull_distribution
{
public:
    // types
typedef RealType result_type;
typedef unspecified param_type;

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

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

    Concepts for Random Number Generation in C++0X (N2781)
26.4 Random number generation

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

```cpp
// 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;
};
```

```cpp
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.

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

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

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

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

26.4.9.3.5 Class template extreme_value_distribution

```cpp
An extreme_value_distribution random number distribution produces random numbers x distributed according to the probability density function

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

```

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

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

5) 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.

Concepts for Random Number Generation in C++0X (N2781)
void reset();

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

// 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.

26.4.9.4 Normal distributions

26.4.9.4.1 Class template normal_distribution

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

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

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

template<
class FloatingPointLike
RealType = double>

requires FloatingPointType<RealType>

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

Concepts for Random Number Generation in C++0X (N2781)
26.4 Random number generation

Numerics library 42

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

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

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

Requires: 0 < stddev.

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

RealType mean() const;

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

RealType stddev() const;

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

26.4.9.4.2 Class template lognormal_distribution

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

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

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

Concepts for Random Number Generation in C++0X (N2781)
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 URNG>
result_type operator()((UniformRandomNumberGenerator URNG& g);

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

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

26.4.9.4.3 Class template chi_squared_distribution

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

Concepts for Random Number Generation in C++0X (N2781)
26.4 Random number generation

// 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 URNG>
result_type operator()(UniformRandomNumberGenerator URNG& g);

// 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.

26.4.9.4.4 Class template cauchy_distribution

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

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

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

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

Concepts for Random Number Generation in C++0X (N2781)
// 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 URNG>
result_type operator()(UniformRandomNumberGenerator& g);
template<class UniformRandomNumberGenerator URNG>
result_type operator()(UniformRandomNumberGenerator& g, const param_type& parm);

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

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

2 Requires: 0 < 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.

26.4.9.4.5 Class template fisher_f_distribution [rand.dist.norm.f]
A fisher_f_distribution random number distribution produces random numbers \( x \geq 0 \) distributed according to the probability density function
\[
p(x|m,n) = \frac{\Gamma((m+n)/2)}{\Gamma(m/2) \Gamma(n/2)} \left( \frac{m}{n} \right)^{m/2} \cdot x^{(m/2)-1} \cdot \left( 1 + \frac{mx}{n} \right)^{-(m+n)/2}.
\]
where \( m \) and \( n \) are positive integers.

template<class FloatingPointLike RealType = double>
requires FloatingPointType<RealType>
class fisher_f_distribution
{
public:
// types

Concepts for Random Number Generation in C++0X (N2781)
26.4 Random number generation

```cpp
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 URNG>
result_type operator()(UniformRandomNumberGenerator URNG& g);
template<class UniformRandomNumberGenerator URNG>
result_type operator()(UniformRandomNumberGenerator URNG& g, 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;
};
```

```cpp
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.
```

26.4.9.4.6 Class template student_t_distribution [rand.dist.norm.t]

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

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

where n is a positive integer.

```cpp
template<class FloatingPointLike RealType = double>
requires FloatingPointType<RealType>
class student_t_distribution
{

Concepts for Random Number Generation in C++0X (N2781)
```
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 URNG>
   result_type operator()(UniformRandomNumberGenerator URNG& g);
   template<class UniformRandomNumberGenerator URNG>
   result_type operator()(UniformRandomNumberGenerator URNG& g, 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);

Requires: 0 < n.

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

int n() const;

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

26.4.9.5 Sampling distributions

26.4.9.5.1 Class template discrete_distribution

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

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

// constructor and reset functions
discrete_distribution();
template<class InputIterator Iter>
    requires Convertible<Iter::value_type, double>
    discrete_distribution(InputIterator firstW, InputIterator lastW);
explicit discrete_distribution(const param_type& parm);
void reset();

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

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

discrete_distribution();

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

template<class InputIterator Iter>
    requires Convertible<Iter::value_type, double>
    discrete_distribution(InputIterator firstW, InputIterator lastW);

Requires:

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

b) If \( \text{firstW} == \text{lastW} \), let the sequence \( w \) have length \( n = 1 \) and consist of the single value \( w_0 = 1 \). Otherwise, \([\text{firstW}, \text{lastW})\) shall form a sequence \( w \) of length \( n > 0 \) and \( \text{firstW} \) 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 \( 0 < S = w_0 + \cdots + w_{n-1} \).

Effects: Constructs a discrete_distribution object with probabilities

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

vector<double> probabilities() const;

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

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

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

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

    // constructor and reset functions
    piecewise_constant_distribution();
    template<class InputIterator IterB, class InputIterator IterW>
        requires Convertible<IterB::value_type, result_type> && Convertible<IterW::value_type, double>
        piecewise_constant_distribution(InputIterator IterB firstB, InputIterator IterB lastB,
                                           InputIterator IterW firstW);
    explicit piecewise_constant_distribution(const param_type& parm);
    void reset();

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

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

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

```c++
template<class InputIterator IterB, class InputIterator IterW>
    requires Convertible<IterB::value_type, result_type> && Convertible<IterW::value_type, double>
    piecewise_constant_distribution(InputIterator IterB firstB, InputIterator IterB lastB,
                                           InputIterator IterW firstW);
```
3 Requires:
   a) `InputIteratorB` shall satisfy the requirements of an input iterator [input.iterator], as shall `InputIteratorW`.
   b) If `firstB == lastB` or the sequence `w` has the length zero,
      (a) let the sequence `w` have length \( n = 1 \) and consist of the single value \( w_0 = 1 \), and
      (b) let the sequence `b` 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

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

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 \( \rho_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n - 1 \).

26.4.9.5.3 Class template `general_pdf_distribution` [rand.dist.samp.genpdf]

A `general_pdf_distribution` random number distribution produces random numbers \( x, x_{\text{min}} \leq x < x_{\text{max}} \), distributed according to the probability density function

\[
p(x | x_{\text{min}}, x_{\text{max}}, \rho) = \rho(x), \text{ for } x_{\text{min}} \leq x < x_{\text{max}}.
\]

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

// constructor and reset functions
general_pdf_distribution();
template<class Callable<auto,result_type> Func>
    requires Convertible<Func::result_type, double>
    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 URNG>
    result_type operator()(UniformRandomNumberGenerator URNG& g);
template<class UniformRandomNumberGenerator URNG>
    result_type operator()(UniformRandomNumberGenerator URNG& g, 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();

Effects: Constructs a general_pdf_distribution object with \( x_{\text{min}} = 0 \) and \( x_{\text{max}} = 1 \) such that \( p(x) = 1 \) for all \( x_{\text{min}} \leq x < x_{\text{max}} \).

template<class Callable<auto,result_type> Func>
    requires Convertible<Func::result_type, double>
    general_pdf_distribution(result_type xmin, result_type xmax, Func pdf);

Requires:

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

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

c) the following relations shall hold:

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

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]

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 \( p(x) = f(x)/z \).
result_type xmin() const;

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

result_type xmax() const;

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

a()
  cauchy_distribution<>, 45
  extreme_value_distribution<>, 41
  uniform_int_distribution<>, 31
  uniform_real_distribution<>, 32
  weibull_distribution<>, 40
alpha()
  gamma_distribution<>, 39
b()
  cauchy_distribution<>, 45
  extreme_value_distribution<>, 41
  uniform_int_distribution<>, 31
  uniform_real_distribution<>, 32
  weibull_distribution<>, 40
base engines
  random number engine adaptor, 10
Bernoulli distributions, 32–36
bernoulli_distribution<>, 32
  constructor, 33
  discrete probability function, 32
  p(), 33
beta()
  gamma_distribution<>, 39
binomial_distribution<>, 33
  constructor, 34
  discrete probability function, 33
  p(), 34
  t(), 34
carry
  subtract_with_carry_engine<>, 19
cauchy_distribution<>, 44
  a(), 45
  b(), 45
  constructor, 45
  probability density function, 44
chi_squared_distribution<>, 43
  constructor, 44
  n(), 44
  probability density function, 43
constructor
  RandomNumberDistribution, 13
  RandomNumberEngine, 9
  RandomNumberEngineAdaptor, 10, 11
  SeedSequence, 15
default_random_engine, 25
densities()
  piecewise_constant_distribution<>, 50
discard()
  RandomNumberEngine, 9
discard_block_engine<>, 21
  constructor, 22
  generation algorithm, 21
  state, 21
  textual representation, 22
  transition algorithm, 21
discrete probability function, 12
  bernoulli_distribution<>, 32
  binomial_distribution<>, 33
  discrete_distribution<>, 47
  geometric_distribution<>, 34
  negative_binomial_distribution<>, 35
  poisson_distribution<>, 36
  uniform_int_distribution<>, 30
discrete_distribution<>
  discrete probability function, 47
discrete_distribution<>, 47
  constructor, 48
discrete_distribution<> 48
probabilities() 48
weights 48
distribution, see random number distribution

engine, see random number engine
engine adaptor, see random number engine adaptor
engines with predefined parameters
default_random_engine 25
knuth_b 25
minstd_rand 24
minstd_rand0 24
mt19937 25
ranlux24 25
ranlux24_base 25
ranlux48 25
ranlux48_base 25
entropy()
random_device 27
exponential_distribution<> 37
constructor 38
lambda() 38
probability density function 37
extreme_value_distribution<> 40
a() 41
b() 41
constructor 41
probability density function 40
fisher_f_distribution<> 45
constructor 46
m() 46
n() 46
probability density function 45
gamma_distribution<> 38
alpha() 39
beta() 39
constructor 39
probability density function 38
general_pdf_distribution<> 50
constructor 51
probability density function 50
xmax() 52
xmin() 52
generate_canonical() 29, 30

generation algorithm
discard_block_engine<> 21
independent_bits_engine<> 22
linear_congruential_engine<> 16
mersenne_twister_engine<> 17
RandomNumberEngine 9
shuffle_order_engine<> 23
subtract_with_carry_engine<> 19
geometric_distribution<> 34
constructor 35
discrete probability function 34
p() 35
independent_bits_engine<> 22
generation algorithm 22
state 22
textual representation 23
transition algorithm 22
interval boundaries
piecewise_constant_distribution<> 49
intervals() 50

knorth_b 25
lambda()
exponential_distribution<> 38
linear_congruential_engine<> 16
constructor 17
generation algorithm 16
modulus 17
state 16
textual representation 17
transition algorithm 16
lognormal_distribution<> 42
constructor 43
m() 43
probability density function 42
s() 43
m()
fisher_f_distribution<> 46
lognormal_distribution<> 43
max()
RandomNumberDistribution 14
mean
normal_distribution<>, 41
poisson_distribution<>, 36
mean()
  normal_distribution<>, 42
  poisson_distribution<>, 37
  student_t_distribution<>, 47
mersenne_twister_engine<>, 17
  constructor, 19
  generation algorithm, 17
  state, 17
  textual representation, 18
  transition algorithm, 17
min()
  RandomNumberDistribution, 14
minstd_rand, 24
minstd_rand0, 24
mt19937, 25

operator()()
  RandomNumberDistribution, 13
operator()()
  random_device, 27
  RandomNumberEngine, 9
  UniformRandomNumberGenerator, 8
operator==()
  RandomNumberDistribution, 13
RandomNumberEngine, 9
RandomNumberEngineAdaptor, 11
operator<<()
  RandomNumberDistribution, 14
RandomNumberEngine, 10
RandomNumberDistribution, 14
RandomNumberEngine, 10
p()  bernoulli_distribution, 33
    binomial_distribution<>, 34
    geometric_distribution<>, 35
    negative_binomial_distribution<>, 36
param()
  RandomNumberDistribution, 13, 14
param()
  seed_seq, 29
  SeedSequence, 16
params
  RandomNumberDistribution, 12
parameters
  random number distribution, 12
piecewise_constant_distribution<>, 49
  constructor, 49
  densities(), 50
  interval boundaries, 49
  intervals(), 50
  probability density function, 49
  weights, 50
Poisson distributions, 36–41
poisson_distribution<>, 36
  constructor, 37
  discrete probability function, 36
  mean, 36
  mean(), 37
probabilities()
  discrete_distribution<>, 48
probability density function, 12
  cauchy_distribution<>, 44
  chi_squared_distribution<>, 43
  exponential_distribution<>, 37
  extreme_value_distribution<>, 40
  fisher_f_distribution<>, 45
  gamma_distribution<>, 38
  general_pdf_distribution<>, 50
  lognormal_distribution<>, 42
  normal_distribution<>, 41
  piecewise_constant_distribution<>, 49

Concepts for Random Number Generation in C++0X (N2781)
student_t_distribution<>: 46
uniform_real_distribution<>: 31
weibull_distribution<>: 39

<random>, 3–7
random number distribution
bernoulli_distribution<>: 32
binomial_distribution<>: 33
chi_squared_distribution<>: 43
corecept, see RandomNumberDistribution
discrete probability function, 12
discrete_distribution<>: 47
exponential_distribution<>: 37
extreme_value_distribution<>: 40
fisher_f_distribution<>: 45
gamma_distribution<>: 38
general_pdf_distribution<>: 50
geometric_distribution<>: 34
lognormal_distribution<>: 42
negative_binomial_distribution<>: 35
normal_distribution<>: 41
parameters, 12
piecewise_constant_distribution<>: 49
poisson_distribution<>: 36
probability density function, 12
student_t_distribution<>: 46
uniform_int_distribution<>: 46
uniform_real_distribution<>: 31

random number distributions
Bernoulli, 32–36
normal, 41–47
Poisson, 36–41
sampling, 47–52
uniform, 30–32

random number engine, 8
corecept, see RandomNumberEngine
linear_congruential_engine<>: 16
mersenne_twister_engine<>: 17
subtract_with_carry_engine<>: 19
with predefined parameters, 24–26
generation algorithm, 9
operator()(), 9
operator==(), 9
operator<<(), 10
operator>>()(), 10
seed(), 9, 11
state, 9
successor state, 9
transition algorithm, 9

AdaptorRandomNumberEngine
operator==(), 11
RandomNumberEngineAdaptor, 10
constructor, 10, 11
ranlux24, 25
ranlux24_base, 25
ranlux48, 25
ranlux48_base, 25
reset()
RandomNumberDistribution, 13
result_type

Concepts for Random Number Generation in C++0X (N2781)
entity characterization based on, 3

\texttt{s()}

\texttt{lognormal_distribution<>}, 43

sampling distributions, 47–52

seed sequence, 15

cancept, see \texttt{SeedSequence}

\texttt{seed()}

\texttt{RandomNumberEngine}, 9, 11

\texttt{seed_seq}

\texttt{constructor}, 28

\texttt{param()}, 29

\texttt{randomize()}, 28

\texttt{size()}, 29

\texttt{SeedSequence}, 14

\texttt{constructor}, 15

\texttt{param()}, 16

\texttt{randomize()}, 15

\texttt{size()}, 16

\texttt{shuffle_order_engine<>}, 23

\texttt{constructor}, 24

\texttt{generation algorithm}, 23

\texttt{state}, 23

\texttt{textual representation}, 24

\texttt{transition algorithm}, 23

\texttt{size()}

\texttt{seed_seq}, 29

\texttt{SeedSequence}, 16

standard deviation

\texttt{normal_distribution<>}, 41

state

\texttt{discard_block_engine<>}, 21

\texttt{independent_bits_engine<>}, 22

\texttt{linear_congruential_engine<>}, 16

\texttt{mersenne_twister_engine<>}, 17

\texttt{RandomNumberEngine}, 9

\texttt{shuffle_order_engine<>}, 23

\texttt{subtract_with_carry_engine<>}, 19

\texttt{stddev()}

\texttt{normal_distribution<>}, 42

\texttt{student_t_distribution<>}, 46

\texttt{constructor}, 47

\texttt{mean()}, 47

\texttt{probability density function}, 46

\texttt{subtract_with_carry_engine<>}, 19

\texttt{carry}, 19

\texttt{constructor}, 20

\texttt{generation algorithm}, 19

\texttt{state}, 19

\texttt{textual representation}, 20

\texttt{transition algorithm}, 19

successor state

\texttt{RandomNumberEngine}, 9

\texttt{t()}

\texttt{binomial_distribution<>}, 34

\texttt{negative_binomial_distribution<>}, 36

textual representation

\texttt{discard_block_engine<>}, 22

\texttt{independent_bits_engine<>}, 23

\texttt{shuffle_order_engine<>}, 24

\texttt{subtract_with_carry_engine<>}, 20

transition algorithm

\texttt{discard_block_engine<>}, 21

\texttt{independent_bits_engine<>}, 22

\texttt{linear_congruential_engine<>}, 16

\texttt{mersenne_twister_engine<>}, 17

\texttt{RandomNumberEngine}, 9

\texttt{shuffle_order_engine<>}, 23

\texttt{subtract_with_carry_engine<>}, 19

unifom distributions, 30–32

uniform random number generator, 7, 12

cancept, see \texttt{UniformRandomNumberGenerator}

\texttt{uniform_int_distribution<>}, 30

\texttt{a()}, 31

\texttt{b()}, 31

\texttt{constructor}, 31

\texttt{discrete probability function}, 30

\texttt{uniform_real_distribution<>}, 31

\texttt{a()}, 32

\texttt{b()}, 32

\texttt{constructor}, 32

\texttt{probability density function}, 31

\texttt{UniformRandomNumberGenerator}, 7

\texttt{operator()()}, 8

\texttt{weibull_distribution<>}, 39

\texttt{a()}, 40

\texttt{b()}, 40

\texttt{constructor}, 40

\texttt{probability density function}, 39

Concepts for Random Number Generation in C++0X (N2781)
weibull_distribution<> 40
weights
discrete_distribution<> 48
piecewise_constant_distribution<> 50

xmax()
general_pdf_distribution<> 52
xmin()
general_pdf_distribution<> 52