This document revises N1932 = Brown, et al.: Random Number Generation in C++0X: A Comprehensive Proposal. It incorporates all known corrections to that paper’s language and typography, including all emendations requested by the Library Working Group during its Berlin meeting (3–7 April, 2006), and also adopts the context of N2009 = Becker: Working Draft, Standard for Programming Language C++. Changes in wording from the previous version of this paper are indicated as either added or deleted text. For brevity, we have indicated in a few places that text has been [moved from here, unchanged]. Minor editorial adjustments in punctuation are not specially formatted.

In a separate document, N2033 = Brown, et al.: Proposal to Consolidate the Subtract-with-Carry Engines, we propose to unify the random number engines `subtract_with_carry_engine<>` and `subtract_with_carry_01_engine<>.` We do so because of these engines’ inherently close relationship: they employ the identical transition algorithm as well as the identical generation algorithm. In our opinion, these engines’ separation was a historical accident based solely on the types (integer- versus real-valued) of the values each was designed to return. We believe it is unnecessary to provide two separate engines whose details are near-identical, and wonder whether it was ever particularly useful to do so.

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.
Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
26 Numerics library

26.4 Random number generation

26.4.1 Requirements

26.4.1.1 General requirements

26.4.1.2 Uniform random number generator requirements

26.4.1.3 Random number engine requirements

26.4.1.4 Random number engine adaptor requirements

26.4.1.5 Random number distribution requirements

26.4.2 Header <random> synopsis

26.4.3 Random number engine class templates

26.4.3.1 Class template linear_congruential_engine

26.4.3.2 Class template mersenne_twister_engine

26.4.3.3 Class template subtract_with_carry_engine

26.4.3.4 Class template subtract_with_carry_01_engine

26.4.4 Random number engine adaptor class templates

26.4.4.1 Class template discard_block_engine

26.4.4.2 Class template shuffle_order_engine

26.4.4.3 Class template xor_combine_engine

26.4.5 Engines with predefined parameters

26.4.6 Class random_device

26.4.7 Function template generate_canonical

26.4.8 Random number distribution class templates

26.4.8.1 Uniform distributions

26.4.8.1.1 Class template uniform_int_distribution

26.4.8.1.2 Class template uniform_real_distribution

26.4.8.2 Bernoulli distributions

26.4.8.2.1 Class bernoulli_distribution

26.4.8.2.2 Class template binomial_distribution

26.4.8.2.3 Class template geometric_distribution

26.4.8.2.4 Class template negative_binomial_distribution
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.4.8.3</td>
<td>Poisson distributions</td>
<td>32</td>
</tr>
<tr>
<td>26.4.8.3.1</td>
<td>Class template poisson_distribution</td>
<td>32</td>
</tr>
<tr>
<td>26.4.8.3.2</td>
<td>Class template exponential_distribution</td>
<td>33</td>
</tr>
<tr>
<td>26.4.8.3.3</td>
<td>Class template gamma_distribution</td>
<td>34</td>
</tr>
<tr>
<td>26.4.8.3.4</td>
<td>Class template weibull_distribution</td>
<td>35</td>
</tr>
<tr>
<td>26.4.8.3.5</td>
<td>Class template extreme_value_distribution</td>
<td>36</td>
</tr>
<tr>
<td>26.4.8.4</td>
<td>Normal distributions</td>
<td>37</td>
</tr>
<tr>
<td>26.4.8.4.1</td>
<td>Class template normal_distribution</td>
<td>37</td>
</tr>
<tr>
<td>26.4.8.4.2</td>
<td>Class template lognormal_distribution</td>
<td>38</td>
</tr>
<tr>
<td>26.4.8.4.3</td>
<td>Class template chi_squared_distribution</td>
<td>39</td>
</tr>
<tr>
<td>26.4.8.4.4</td>
<td>Class template cauchy_distribution</td>
<td>40</td>
</tr>
<tr>
<td>26.4.8.4.5</td>
<td>Class template fisher_f_distribution</td>
<td>41</td>
</tr>
<tr>
<td>26.4.8.4.6</td>
<td>Class template student_t_distribution</td>
<td>42</td>
</tr>
<tr>
<td>26.4.8.5</td>
<td>Sampling distributions</td>
<td>43</td>
</tr>
<tr>
<td>26.4.8.5.1</td>
<td>Class template discrete_distribution</td>
<td>43</td>
</tr>
<tr>
<td>26.4.8.5.2</td>
<td>Class template piecewise_constant_distribution</td>
<td>44</td>
</tr>
<tr>
<td>26.4.8.5.3</td>
<td>Class template general_pdf_distribution</td>
<td>46</td>
</tr>
</tbody>
</table>
### List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Uniform random number generator requirements</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Random number engine requirements</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Random number engine adaptor requirements</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Random number distribution requirements</td>
<td>6</td>
</tr>
</tbody>
</table>
26 Numerics library [lib.numerics]

26.4 Random number generation [lib.random.numbers]

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

2 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 \( \text{std}::\text{tr1}::\text{bind}(d, e) \). — end note]

3 Each of the entities specified via this subclause has an associated arithmetic type [basic.fundamental] identified as result_type. With \( T \) as the result_type thus associated with such an entity, that entity is characterized

   a) as boolean or equivalently as boolean-valued, if \( T \) is bool;

   b) otherwise as integral or equivalently as integer-valued, if \( \text{numeric_limits<T>::is_integer} \) is true;

   c) otherwise as floating or equivalently as real-valued.

   If integer-valued, an entity may optionally be further characterized as signed or unsigned, according to \( T \).

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

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

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

   b) the operator lshift, whose result is always taken modulo \( 2^w \).

26.4.1 Requirements [lib.rand.req]

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

1 The effect of instantiating a template

   a) that has a template type parameter named UniformRandomNumberGenerator is undefined unless the corresponding template argument satisfies the requirements of uniform random number generator [26.4.1.2].
Random number generation

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

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

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

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

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

### 26.4.1.2 Uniform random number generator requirements

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

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

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

**Table 1: Uniform random number generator requirements**

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X::result_type</code></td>
<td><code>T</code></td>
<td><code>T</code> is an arithmetic type [basic.fundamental] other than <code>bool</code>.</td>
<td>compile-time</td>
</tr>
<tr>
<td><code>u()</code></td>
<td><code>T</code></td>
<td>If <code>X</code> is integral, returns a value in the closed interval <code>[X::min, X::max]</code>; otherwise, returns a value in the open interval <code>(0, 1)</code>.</td>
<td>amortized constant</td>
</tr>
<tr>
<td><code>X::min</code></td>
<td><code>T</code>, if <code>X</code> is integral; otherwise <code>int</code>.</td>
<td>If <code>X</code> is integral, denotes the least value potentially returned by <code>operator()</code>; otherwise denotes 0.</td>
<td>compile-time</td>
</tr>
<tr>
<td><code>X::max</code></td>
<td><code>T</code>, if <code>X</code> is integral; otherwise <code>int</code>.</td>
<td>If <code>X</code> is integral, denotes the greatest value potentially returned by <code>operator()</code>; otherwise denotes 1.</td>
<td>compile-time</td>
</tr>
</tbody>
</table>

### 26.4.1.3 Random number engine requirements

A class `X` that satisfies the requirements of a uniform random number generator [26.4.1.2] also satisfies the requirements

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

2) It is intended that this list be augmented with `unsigned long long` if and when this type is incorporated into the Working Paper.
of a random number engine if the expressions shown in table 2 are valid and have the indicated semantics, and if \( X \) also satisfies all other requirements of this section 26.4.1.3. In that table and throughout this section 26.4.1.3,

a) \( T \) is the type named by \( X \)'s associated \texttt{result\_type};

b) \( t \) is a value of \( T \);

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

d) \( s \) is a value of integral type;

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

f) \( \texttt{os} \) is an lvalue of the type of some class template specialization \texttt{basic\_ostream\lt charT, traits\gt }; and

g) \( \texttt{is} \) is an lvalue of the type of some class template specialization \texttt{basic\_istream\lt charT, traits\gt }; where \texttt{charT} and \texttt{traits} are constrained according to [lib.strings] and [lib.input.output].

A random number engine \texttt{object} \( x \) has at any given time a state \( x_i \) for some integer \( i \geq 0 \). Upon successful instantiation-\texttt{construction}, a random number engine \( x \) has an initial state \( x_0 \). An engine’s state may be established by invoking its \texttt{constructor}, seed \texttt{member function}, \texttt{operator=} , or a suitable \texttt{operator\texttt{>>}}.

The specification of each random number engine defines the size of its state in multiples of the size of its \texttt{result\_type}, given as an integral constant expression. The specification of each random number engine also defines

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

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

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X() )</td>
<td>—</td>
<td>Creates an engine with the same initial state as all other default-constructed engines of type ( X ).</td>
<td>( \Theta(\text{size of state}) )</td>
</tr>
<tr>
<td>( X(s) )</td>
<td>—</td>
<td>Creates an engine with initial state determined by \texttt{static_cast\lt unsigned long\gt (s)}.</td>
<td>( \Theta(\text{size of state}) )</td>
</tr>
<tr>
<td>( X(g) )</td>
<td>—</td>
<td>Creates an engine with initial state determined by the results of successive invocations of ( g ). \texttt{Throws what and when g throws.}</td>
<td>( \Theta(\text{size of state}) )</td>
</tr>
</tbody>
</table>

| \( \texttt{u.seed()} \) | \texttt{void} | post: \( u == X() \) | same as \( X() \) |
| \( \texttt{u.seed(s)} \)   | \texttt{void} | post: \( u == X(s) \) | same as \( X(s) \) |

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>u.seed(g)</td>
<td>void</td>
<td>post: If <code>g</code> does not throw, <code>u == X(g)</code> (u == v), where the state of <code>v</code> is as if constructed by <code>X(g)</code>. Otherwise, the exception is rethrown and the engine's state is deemed invalid. Thereafter, further use of <code>u</code> is undefined except for destruction or invoking a function that establishes a valid state.</td>
<td>same as <code>X(g)</code></td>
</tr>
<tr>
<td>u()</td>
<td>T</td>
<td>Sets the state to (u_{i+1} = TA(u_i)) and returns (GA(u_i)).</td>
<td>amortized constant</td>
</tr>
<tr>
<td><code>x == y</code></td>
<td>bool</td>
<td>With (S_x) and (S_y) as the infinite sequences of values that would be generated by repeated calls to <code>x()</code> and <code>y()</code>, respectively, returns <code>true</code> if (S_x = S_y); returns <code>false</code> otherwise.</td>
<td>(O(\text{size of state}))</td>
</tr>
<tr>
<td><code>x != y</code></td>
<td>bool</td>
<td>!(<code>x == y</code>)</td>
<td>(O(\text{size of state}))</td>
</tr>
<tr>
<td><code>os &lt;&lt; x</code></td>
<td>reference to the type of <code>os</code></td>
<td>With <code>os</code>.fmtflags set to `ios_base::dec</td>
<td>ios_base::fixed</td>
</tr>
<tr>
<td><code>is &gt;&gt; v</code></td>
<td>reference to the type of <code>is</code></td>
<td>Sets <code>v</code>'s state as determined by reading its textual representation from <code>is</code>. If bad input is encountered, ensures that <code>v</code>'s state is unchanged by the operation and calls <code>is.setstate(ios::failbit)</code> (which may throw <code>ios::failure [lib.iostate.flags]</code>). pre: The textual representation was previously written using an <code>os</code> whose imbued locale and whose type's template specialization arguments <code>charT</code> and <code>traits</code> were the same as those of <code>is</code>. post: The <code>is</code>.fmtflags are unchanged.</td>
<td>(O(\text{size of state}))</td>
</tr>
</tbody>
</table>
X shall satisfy the requirements of uniform random number generator [26.4.1.2] as well as of CopyConstructible [lib.copyconstructible] and of Assignable [lib.container.requirements]. Copy construction and assignment shall each be of complexity $O(\text{size of state})$.

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

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 $v$.

### 26.4.1.4 Random number engine adaptor requirements

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

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

1) $B_i$ is the type of the $i^{th}$ of $X$’s base engines, $1 \leq i \leq n$; and

2) $b_i$ is a value of $B_i$.

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

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X::base_i_type$</td>
<td>$B_i$</td>
<td>—</td>
<td>compile time</td>
</tr>
<tr>
<td>$X::base_i()$</td>
<td>$\text{const } B_i$ &amp;</td>
<td>Returns a reference to $b_i$.</td>
<td>constant</td>
</tr>
</tbody>
</table>

$X$ shall satisfy the requirements of random number engine [26.4.1.3], subject to the following interpretations of those requirements:

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

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

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

4) The textual representation of $X$ includes, in turn, the textual representation of each of its base engines.
e) When \( X::\text{seed} \) is invoked with no arguments, each of \( X \)'s base engines is \texttt{seeded constructed}, in turn, as if by its respective default constructor. When \( X::\text{seed} \) is invoked with an unsigned long value \( s \), each of \( X \)'s base engines is \texttt{seeded constructed}, in turn, with the next available value from the list \( s, s+1, \ldots \). When \( X::\text{seed} \) is invoked with a zero-argument function object, each of \( X \)'s base engines is \texttt{seeded constructed}, in turn, with that function object as argument. [Note: This permits the function object to accumulate side effects. — end note]

f) The equality operator, applied to two operands \( a_1 \) and \( a_2 \) of identical engine adaptor type, returns true if and only if each base engine of \( a_1 \) compares equal, in turn, to the corresponding base engine of \( a_2 \).

\( X \) shall have one additional constructor with as many arguments as \( X \) has base engines. These parameters’ types shall correspond to the types of the base engines' \( n \) or more parameters such that the type of parameter \( i \), \( 1 \leq i \leq n \), is \texttt{const B_i&} and such that all remaining parameters, if any, have default values. The constructor shall construct \( X \), initializing each of its base engines, in turn, with a copy of the value of the corresponding argument, in no way modifying any of the argument values.

26.4.1.5 Random number distribution requirements

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

\begin{itemize}
  \item a) \( T \) is the type named by \( X \)'s associated \texttt{result_type};
  \item b) \( P \) is the type named by \( X \)'s associated \texttt{param_type};
  \item c) \texttt{u and x are} is a values of \( X \) and \( x \) is a (possibly \texttt{const}) value of \( X \);
  \item d) \texttt{glb} and \texttt{1ub} are values of \( T \) respectively corresponding to the greatest lower bound and the least upper bound on the values potentially returned by \( u \)'s \texttt{operator()}\), as determined by the current values of \( u \)'s parameters;
  \item e) \( p \) is a value of \( P \);
  \item f) \( e \) is an \texttt{lvalue} of an arbitrary type that satisfies the requirements of a uniform random number generator [26.4.1.2];
  \item g) \( os \) is an \texttt{lvalue} of the type of some class template specialization \texttt{basic_ostream<charT, traits>}; and
  \item h) \( is \) is an \texttt{lvalue} of the type of some class template specialization \texttt{basic_istream<charT, traits>};
\end{itemize}

where \texttt{charT} and \texttt{traits} are constrained according to [lib.strings] and [lib.input.output].

The specification of each random number distribution identifies an associated mathematical \textit{probability density function} \( p(z) \) or an associated \textit{discrete probability function} \( P(z_i) \). Such functions are typically expressed using certain externally-supplied quantities identified as the \textit{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.

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X::\text{result_type} )</td>
<td>( T )</td>
<td>( T ) is an arithmetic type.</td>
<td>compile-time</td>
</tr>
<tr>
<td>expression</td>
<td>return type</td>
<td>pre/post-condition</td>
<td>complexity</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>X::param_type P</td>
<td>P</td>
<td>P is constructible from the identical values used in the construction of any value of X.</td>
<td>compile-time</td>
</tr>
<tr>
<td>X(p)</td>
<td>—</td>
<td>Creates a distribution whose behavior is indistinguishable from that of a distribution newly constructed directly from the values used to construct p.</td>
<td>same as p’s construction</td>
</tr>
<tr>
<td>u.reset()</td>
<td>void</td>
<td>Subsequent uses of u do not depend on values produced by e prior to invoking reset.</td>
<td>constant</td>
</tr>
<tr>
<td>u.param()</td>
<td>P</td>
<td>Returns a value that could have been used to construct u in its initial state such that X(p).param() == p.</td>
<td>no worse than the complexity of X(p)</td>
</tr>
<tr>
<td>u.param(p)</td>
<td>void</td>
<td>post: u.param() == p.</td>
<td>no worse than the complexity of X(p)</td>
</tr>
<tr>
<td>u(e)</td>
<td>T</td>
<td>The sequence of numbers returned by successive invocations with the same object e is randomly distributed according to the associated p(z</td>
<td>{p}) or P(z</td>
</tr>
<tr>
<td>u(e,p)</td>
<td>T</td>
<td>Returns the same result as would v(e) where v had been freshly constructed with argument p. The sequence of numbers returned by successive invocations with the same objects e and p is randomly distributed according to the associated p(z</td>
<td>{p}) or P(z</td>
</tr>
<tr>
<td>x.min()</td>
<td>T</td>
<td>Returns glb.</td>
<td>constant</td>
</tr>
<tr>
<td>x.max()</td>
<td>T</td>
<td>Returns lub.</td>
<td>constant</td>
</tr>
<tr>
<td>os &lt;&lt; x</td>
<td>reference to the type of os</td>
<td>Writes to os a textual representation for the parameters and the additional internal data of x. post: The os.fmtflags and fill character are unchanged.</td>
<td>O(size of state)</td>
</tr>
</tbody>
</table>

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
is >> u references to the type of is
Restores from is the parameters and additional internal data of u. If bad input is encountered, ensures that u is unchanged by the operation and calls is.setstate(ios::failbit) (which may throw ios::failure [lib.iostate.flags]).
pre: is provides a textual representation that was previously written using an os whose imbued locale and whose type’s template specialization arguments charT and traits were the same as those of is.
post: The is.fmtflags are unchanged.

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

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

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

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

7 P shall satisfy the requirements of CopyConstructible, Assignable, and EqualityComparable [lib.equalitycomparable]. Copy construction and assignment shall each be of complexity O(number of values used in the source’s construction).

8 For each of the constructors of X taking arguments corresponding to parameters of the distribution, P shall have a corresponding constructor subject to the same requirements and taking arguments identical in number, type, and default values. Moreover, for each of the member functions of X that return values corresponding to parameters of the distribution, P shall have a corresponding member function with the identical name, type, and semantics.

26.4.2 Header <random> synopsis

namespace std {

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
// [26.4.3.2] Class template mersenne_twister_engine
template <class UIntType, int w, int n, int m, int r,
         UIntType a, int u, int s, UIntType b, int t, UIntType c, int l>
  class mersenne_twister_engine;

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

// [26.4.3.4] Class template subtract_with_carry_01_engine
template <class RealType, int w, int s, int r>
  class subtract_with_carry_01_engine;

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

// [26.4.4.2] Class template shuffle_order_engine
template <class Engine, int k>
  class shuffle_order_engine;

// [26.4.4.3] Class template xor_combine_engine
template <class Engine1, int s1, class Engine2, int s2>
  class xor_combine_engine;

// [26.4.5] Engines with predefined parameters
typedef see below minstd_rand0;
typedef see below minstd_rand;
typedef see below mt19937;
typedef see below ranlux_base_01;
typedef see below ranlux64_base_01;
typedef see below ranlux3;
typedef see below ranlux4;
typedef see below ranlux3_01;
typedef see below ranlux4_01;
typedef see below knuth_b;

// [26.4.6] Class random_device
class random_device;

// [26.4.7] Function template generate_canonical
template<class result_type, class UniformRandomNumberGenerator>
  result_type generate_canonical( UniformRandomNumberGenerator & g );

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

// [26.4.8.1.2] Class template uniform_real_distribution
template <class RealType = double>
  class uniform_real_distribution;
26.4 Random number generation

class uniform_real_distribution;

// [26.4.8.2.1] Class bernoulli_distribution
class bernoulli_distribution;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

// [26.4.8.5.3] Class template general_pdf_distribution
template <class RealType = double>
    class general_pdf_distribution;

} // namespace std

26.4.3 Random number engine class templates

Except where specified otherwise, the complexity of all functions specified in the following sections is constant. Except for constructors and for the seed functions that take a zero argument function object as required by table 2, no function described in this section 26.4.3 throws an exception. Except where specified otherwise, the class templates specified in this section 26.4.3 satisfy the requirements of random number engine [26.4.1.3]. Descriptions are provided here only for operations on the engines that are not described in those requirements or for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment operators, and for equality and inequality operators are not shown in the synopsis.

26.4.3.1 Class template linear_congruential_engine

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

template <class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine
{
public:
    // types
    typedef UIntType result_type;

    // parameter values and engine characteristics
    static const UIntType multiplier = a;
26.4 Random number generation

```cpp
static const UIntType increment = c;
static const UIntType modulus = m;
static const result_type min = c == 0u ? 1u: 0u;
static const result_type max = m - 1u;
static const unsigned long default_seed = 1uL;

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

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

1 The template parameter UIntType shall denote an unsigned integral type large enough to store values as large as \( m - 1 \).

If the template parameter \( m \) is 0, the modulus \( m \) used throughout this section 26.4.3.1 is \( \text{numeric_limits<UIntType>::max()} \) plus 1. [Note: The result is \textit{not} representable as a value of type UIntType. — end note] Otherwise, the following relations shall hold: \( a < m \) and \( c < m \).

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

```cpp
explicit linear_congruential_engine(unsigned long s = default_seed);
```

4 Effects: Constructs a \texttt{linear_congruential_engine} object as if by invoking \texttt{seed(s)}.

```cpp
void seed(unsigned long s = default_seed);
```

5 Effects: If \( c \mod m \) is 0 and \( s \mod m \) is 0, sets the engine’s state to 1, otherwise sets the engine’s state to \( s \mod m \).

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

5 Effects: Constructs a \texttt{linear_congruential_engine} object. If \( \gamma = g() \mod m \), if \( c \mod m \) is 0 and \( g() \mod m \) is 0, sets the engine’s state to 1, else sets the engine’s state to \( g() \mod m \).

6 Complexity: Exactly one invocation of \( g \).

26.4.3.2 Class template mersenne_twister_engine

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

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

The state transition is performed as follows:

---

3) The name of this engine refers, in part, to a property of its period: For properly-selected values of the parameters, the period is closely related to a large Mersenne prime number.
a) Concatenate the upper $w - r$ bits of $X_{i - n}$ with the lower $r$ bits of $X_{i + m - n}$ to obtain an unsigned integer value $Y$.
b) With $\alpha = a \cdot (Y \text{ bitand } 1)$, set $X_i$ to $X_{i + m - n} \text{ xor } (Y \text{ rshift } 1) \text{ xor } \alpha$.

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 $z_1 = X_i \text{ xor } (X_i \text{ rshift } u)$.
b) Let $z_2 = z_1 \text{ xor } ((z_1 \text{ lshift }_w s) \text{ bitand } b)$.
c) Let $z_3 = z_2 \text{ xor } ((z_2 \text{ lshift }_w t) \text{ bitand } c)$.
d) Let $z_4 = z_3 \text{ xor } (z_3 \text{ rshift } l)$.

The template parameter `UIntType` shall denote an unsigned integral type large enough to store values up to $2^w - 1$. Also, the following relations shall hold: $1 \leq m \leq n$, $0 \leq u, s, t, l \leq w$, $0 \leq a, b, c \leq 2^w - 1$, $1 \leq m \leq n$; $0 \leq r, u, s, t, l \leq w$; $0 \leq a, b, c \leq 2^w - 1$.

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

explicit mersenne_twister_engine(unsigned long value = 5489UL default_seed);

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

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

Complexity: \( \mathcal{O}(n) \).

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

Effects: Constructs a mersenne_twister_engine object. Given the values \( z_0, \ldots, z_{n-1} \) obtained by successive invocations of \( g \), sets \( X_{-n}, \ldots, X_{-1} \) to \( z_0 \mod 2^w, \ldots, z_{n-1} \mod 2^w \), respectively.

Complexity: Exactly \( n \) invocations of \( g \).

void seed(unsigned long value);

effects: Sets \( X_{-n} \) to value \mod 2^w. Then, iteratively for \( i = 1 - n, \ldots, -1 \), sets \( X_{i-n} \) to [formula moved, unchanged, to constructor above].

26.4.3.3 Class template subtract_with_carry_engine

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

The transition algorithm is a modular linear function of the form \( TA(x_i) = (a \cdot x_i) \mod p \), where \( p \) is of the form \( m' - m' + 1 \) and \( a = p - \frac{m-1}{p} \). 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.

The generation algorithm yields the last value of \( Y \mod m \) produced as a result of advancing the engine’s state as described above.

template <class IntType, IntType m, int s, int r>
class subtract_with_carry_engine
{

public:

    // types
    typedef IntType result_type;

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
static const unsigned long default_seed = 19780503uL;

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

// generating functions
result_type operator()();

4 The template parameter IntType shall denote a signed integral type large enough to store values up to $m$.
5 The following relations shall hold: $0 < s < r$.
6 The textual representation consists of the values of $X_{i-r}, \ldots, X_{i-1}$ and $c$, in that order.

explicit subtract_with_carry_engine(unsigned long value = 19780503uL);

Effects: Constructs a subtract_with_carry_engine object as if by invoking seed(value) the constructor subtract_with_carry_engine(g) had been invoked, where $g$ had been freshly constructed as if by the following definition:

$$\text{linear_congruential_engine<unsigned long,40014,0,2147483563> g(value == 0uL ? default_seed : value);}$$

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

Effects: Constructs a subtract_with_carry_engine object.

With $n = \lfloor (\log_2 m + 31)/32 \rfloor$ and given the values $z_0, \ldots, z_{n(r-1)}$ obtained by successive invocations of $g$ taken modulo $2^{32}$, sets $X_{i-r}, \ldots, X_{i-1}$ to $(z_0 + z_1 \cdot 2^{32} + \cdots + z_{n-1} \cdot 2^{32(n-1)}) \mod m$, respectively. If $X_{i-1}$ is then 0, sets $c$ to 1; otherwise sets $c$ to 0.

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

void seed(unsigned long value = 19780503uL);

Effects: If value is 0, ... [algorithm moved, unchanged, to constructor above].

26.4.3.4 Class template subtract_with_carry_01_engine

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

The transition algorithm is a modular linear function of the form $TA(x_i) = (a \cdot x_i) \mod p$, where $p$ is of the form $2^wr - 2^w + 1$ and $a = p - \frac{p-1}{r}$. The state transition is performed as follows:

a) Let $Y = X_{i-1} - X_{i-r} - c$.

b) Set $X_i$ to $Y \mod 2^w$. Set $c$ to 1 if $Y < 0$, otherwise set $c$ to 0.
26.4 Random number generation

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

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

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

        // parameter values and engine characteristics
        static const int word_size = w;
        static const int short_lag = s;
        static const int long_lag = r;
        static const int min = 0;
        static const int max = 1;
        static const unsigned long default_seed = 19780503uL;

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

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

4 The following relations shall hold: \( 0 < s < r \) and \( w < -\text{numeric_limits<RealType>::min_exponent}-2 \). [Note: The latter relation ensures that \( \epsilon \), used above, is representable as a non-zero value of result_type. — end note]

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

```cpp
explicit subtract_with_carry_01_engine(unsigned long value = 19780503uLdefault_seed);
```

6 Effects: Constructs a subtract_with_carry_01_engine object as if by invoking `seed(value)` the constructor `subtract_with_carry_01_engine(g)` had been invoked, where `g` had been freshly constructed as if by the following definition:

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

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

Effects: If \( \text{value} \) is 0, ... [algorithm moved, unchanged, to constructor above].

complexity: Exactly \( n \cdot r \) invocations of \( g \).

```cpp
template <class Engine, int p, int r>
class discard_block_engine
{
public:
  // types
  typedef Engine base_type;
  typedef typename base_type::result_type result_type;

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

  // constructors and seeding functions
  discard_block_engine();
  explicit discard_block_engine(const base_type& urng);
};
```

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
explicit discard_block_engine(unsigned long s);
template <class Gen> explicit discard_block_engine(Gen& g);
void seed();
void seed(unsigned long s);
template <class Gen> void seed(Gen& g);

// generating functions
result_type operator()();

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

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

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 26.4.1.4, each constructor that is not a copy constructor sets \( n \) to 0.

discard_block_engine();
effects: Constructs a discard_block_engine object. To construct the subobject \( e \), invokes the default constructor of base_type. Sets \( n \) to 0.

discard_block_engine(const base_type& urng);
effects: Constructs a discard_block_engine object. Initializes \( e \) with a copy of \( urng \). Sets \( n \) to 0.

discard_block_engine(unsigned long s);
effects: Constructs a discard_block_engine object. To construct the subobject \( e \), invokes the base_type\( (e) \) constructor. Sets \( n \) to 0.

template <class Gen> explicit discard_block_engine(Gen& g);
effects: Constructs a discard_block_engine object. To construct the subobject \( e \), invokes the base_type\( (e) \) constructor. Sets \( n \) to 0.

void seed();
effects: Invokes \( e\).seed() and sets \( n \) to 0.

void seed(unsigned long s);
effects: Invokes \( e\).seed(s) and sets \( n \) to 0.

Returns: A reference to \( e \).

26.4.4.2 Class template shuffle_order_engine

A shuffle_order_engine random number engine adaptor produces the same random numbers that are produced by

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
some base engine \( e \), but delivers them in a different sequence. The state \( x_i \) of a \texttt{shuffle
d_order_engine} engine adaptor object \( x \) consists of the state \( e_i \) of its base engine \( e \), an
additional value \( Y \) of the type delivered by \( e \), and an additional
sequence \( V \) of \( k \) values also of the type delivered by \( e \). The size of the state is the size of \( e \)'s state plus \( k + 1 \).

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

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

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

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

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

    // generating functions
    result_type operator()();

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

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

4 The following relation shall hold: \( 1 \leq k \).

5 The textual representation consists of the textual representation of \( e \), followed by the \( k \) values of \( V \), followed by the value of \( Y \).
In addition to its behavior pursuant to section 26.4.1.4, each constructor that is not a copy constructor initializes \( V[0], \ldots, V[k-1] \) and \( Y \), in that order, with values returned by successive invocations of \( e() \).

**shuffle_order_engine();**

effects: Constructs a \( \text{shuffle} \_\text{order} \_\text{engine} \) object. To construct the subobject \( e \), invokes the default constructor of \( \text{base} \_\text{type} \). Initializes \( V[0], \ldots, V[k-1] \) and \( Y \), in that order, with values obtained from successive invocations of \( e() \).

**shuffle_order_engine(const base\_type& urng);**

effects: Constructs a \( \text{shuffle} \_\text{order} \_\text{engine} \) object. Initializes \( e \) with a copy of \( \text{urng} \). Initializes \( V[0], \ldots, V[k-1] \) and \( Y \) as described above.

**shuffle_order_engine(unsigned long s);**

effects: Constructs a \( \text{shuffle} \_\text{order} \_\text{engine} \) object. To construct the subobject \( e \), invokes the \( \text{base} \_\text{type} \) constructor. Initializes \( V[0], \ldots, V[k-1] \) and \( Y \) as described above.

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

effects: Constructs a \( \text{shuffle} \_\text{order} \_\text{engine} \) object. To construct the subobject \( e \), invokes the \( \text{base} \_\text{type} \) constructor. Initializes \( V[0], \ldots, V[k-1] \) and \( Y \) as described above.

**void seed();**

effects: Invokes \( e \_\text{seed}() \) and initializes \( V[0], \ldots, V[k-1] \) and \( Y \), as described above.

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

effects: Invokes \( e \_\text{seed}(s) \) and initializes \( V[0], \ldots, V[k-1] \) and \( Y \), as described above.

**const base\_type& base() const;**

Returns: A reference to \( e \).

### 26.4.4.3 Class template xor\_combine\_engine

An \( \text{xor}\_\text{combine}\_\text{engine} \) random number engine adaptor produces random numbers from two integer-valued base engines \( e1 \) and \( e2 \) by merging their left-shifted random values via bitwise exclusive-or. The state \( x_i \) of a \( \text{xor}\_\text{combine}\_\text{engine} \) engine adaptor object \( x \) consists of the states \( e1_i \) and \( e2_i \) of its base engines. The size of the state is the size of the state of \( e1 \) plus the size of the state of \( e2 \).

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

The generation algorithm is \( \text{GA}(x_i) = (e1() \mid \text{lshift\_w} s1) \text{xor} (e2() \mid \text{lshift\_w} s2) \), where \( w \) denotes the value of \( \text{numeric}\_\text{limits<result\_type>::digits} \).

```
template <class Engine1, int s1, class Engine2, int s2>
class xor\_combine\_engine
{
  public:
    // types
    typedef Engine1 base1\_type;
    typedef Engine2 base2\_type;
    typedef see below\_type result\_type;

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

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

// generating functions
result_type operator()();

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

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

4 The following relations shall hold: \( s_1 \geq s_2 \geq 0 \).

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

a) While two shift values (template parameters \( s_1 \) and \( s_2 \)) are provided for simplicity of interface, it is advisable that at most one of these values be nonzero. (If both \( s_1 \) and \( s_2 \) are nonzero then the low bits will always be zero.)

b) It is also advisable for the unshifted base engine’s \( \text{max} \) to be \( 2^n - 1 - \text{min} \) for some non-negative integer \( n \), and for the shift applied to the other base engine to be no greater than that \( n \).

— end note]

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

7 With \( w \) as above, and given the unsigned integer values

a) \( m_1 = (\text{Engine1}::\text{max} - \text{Engine1}::\text{min}) \ll \text{shift}_w \) \((s_1 - s_2)\),

b) \( m_2 = \text{Engine2}::\text{max} - \text{Engine2}::\text{min} \),

c) \( A = m_1 \& m_2 \),

d) \( B = m_1 \| m_2 \), and

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
the value of the member max is \((C \text{ bitor } B) \text{ lshift}_w(s1 - s2)\).

The textual representation consists of the textual representation of \(e1\) followed by the textual representation of \(e2\).

```
xor_combine_engine();
```

**effects:** Constructs a `xor_combine_engine` object. To construct each of the subobjects \(e1\) and \(e2\), invokes their respective default constructors.

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

**effects:** Constructs a `xor_combine_engine` object. Initializes the subobject \(e1\) with a copy of \(urng1\) and then initializes the subobject \(e2\) with a copy of \(urng2\).

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

**effects:** Constructs a `xor_combine_engine` object. Initializes the subobject \(e1\) by invoking the \(e1(s)\) constructor, and then initializes the subobject \(e2\) by invoking the \(e2(s+1)\) constructor. [Note: If both \(e1\) and \(e2\) are of the same type, both base engines should not be initialized with the same seed. — end note]

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

**effects:** Constructs a `xor_combine_engine` object. Initializes the subobject \(e1\) by invoking the \(e1(g)\) constructor, and then initializes the subobject \(e2\) by invoking the \(e2(g)\) constructor.

```
void seed();
```

**effects:** Invokes \(e1.seed()\) and \(e2.seed()\).

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

**Returns:** A reference to \(e1\).

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

**Returns:** A reference to \(e2\).

### 26.4.5 Engines with predefined parameters

[lib.rand.peref]

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

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

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

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

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

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
typedef subtract_with_carry_01_engine<float, 24, 10, 24>
ranlux_base_01;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type ranlux_base_01 shall produce the value 7937952 \cdot 2^{-24}.

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

Required behavior: The 10000th consecutive invocation of a default-constructed object of type ranlux64_base_01 shall produce the value 192113843633948 \cdot 2^{-48}.

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

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

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

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

typedef discard_block_engine<ranlux_base_01, 223, 24>
ranlux3_01;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type ranlux3_01 shall produce the value 5957620 \cdot 2^{-24}.

typedef discard_block_engine<ranlux_base_01, 389, 24>
ranlux4_01;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type ranlux4_01 shall produce the value 8587295 \cdot 2^{-24}.

typedef shuffle_order_engine<minstd_rand0, 256>
knuth_b;

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

26.4.6 Class random_device

A random_device uniform random number generator produces non-deterministic random numbers. It satisfies the requirements of uniform random number generator [lib.rand.req.urng]. Descriptions are provided here only for operations that are not described in those requirements or for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment operators, and for equality and inequality operators are not shown in the synopsis.
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 const result_type min = see below;
    static const result_type max = see below;

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

    // generating functions
    result_type operator()();

    // property functions
    double entropy() const;

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

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

**Effects:** Constructs a `random_device` non-deterministic uniform random number engine generator object. The semantics and default value of the token parameter are implementation-defined.\(^4\)

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

```cpp
result_type min() const;
returns numeric_limits<result_type>::min().
result_type max() const;
returns numeric_limits<result_type>::max().
```

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

**Returns:** An If the implementation employs a random number engine, returns zero. Otherwise, returns an entropy estimate\(^5\) for the random numbers returned by `operator()`, in the range \(\min() \) to \(\log_2(\max() + 1)\). \([\text{Note: A deterministic random number generator (e.g., a random number engine) has entropy 0.} - \text{end note}]\)

---

\(^4\) The parameter is intended to allow an implementation to differentiate between different sources of randomness.

\(^5\) If a device has \(n\) states whose respective probabilities are \(P_0, \ldots, P_{n-1}\), the device entropy \(S\) is defined as \(S = -\sum_{i=0}^{n-1} P_i \log P_i\).
7          Throws: Nothing.

    result_type operator()();

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

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

26.4.7 Function template generate_canonical [lib.rand.canonical]

1          Each function instantiated from the template described in this section 26.4.7 maps the result of a single invocation of a supplied uniform random number generator to one member of \( \mathcal{I} \) (described below) such that, if the values produced by the generator are uniformly distributed, the instantiation’s results are distributed as uniformly as possible according to the uniformity requirements described below.

2          For purposes of this section 26.4.7, let \( \mathcal{I} \) consist of all values \( t \) of type result_type such that:

3          a) If result_type is a floating-point type [basic.fundamental], result_type(0) < \( t \) < result_type(1).

3          b) If result_type is a signed or unsigned integral type [basic.fundamental], numeric_limits<result_type>::min() \leq \( t \) \leq numeric_limits<result_type>::max().

3          [ Note: Obtaining a value in \( \mathcal{I} \) 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 ]

4          template<class result_type, class UniformRandomNumberGenerator>

4          result_type generate_canonical(UniformRandomNumberGenerator & g);

5          Returns: A value from \( \mathcal{I} \), subject to the following required behavior.

6          Required behavior: Let \( |\mathcal{I}| \) denote the number of distinct values in \( \mathcal{I} \), and let \( |g| \) denote the number of distinct values that \( g \) is capable of producing. Finally, let \( x \) be the value resulting from the invocation of \( g \).

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

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

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

6          Complexity: Exactly one invocation of \( g \).
26.4 Random number generation

26.4.8 Random number distribution class templates

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

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 \( P(z) \) specified in this section is 0 everywhere outside its stated domain.

26.4.8.1 Uniform distributions

26.4.8.1.1 Class template uniform_int_distribution

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

\[
P(i|a,b) = \frac{1}{(b - a + 1)}
\]

where \( a \) and \( b \) are the parameters of the distribution.

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

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

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

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
explicit uniform_int_distribution(IntType a = 0, IntType b = numeric_limits<IntType>::max());

Requires: $a \leq b$.

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

result_type a() const;

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

result_type b() const;

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

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

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

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

where $a$ and $b$ are the parameters of the distribution.

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

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

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

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
explicit uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);

Requires: a ≤ b.

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

result_type a() const;

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

result_type b() const;

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

### 26.4.8.2 Bernoulli distributions

**Class bernoulli_distribution**

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

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

where \( p \) is the parameter of the distribution.

```cpp
class bernoulli_distribution
{
public:
    // types
    typedef bool result_type;
    typedef unspecified param_type;
    // constructors and reset functions
    explicit bernoulli_distribution(double p = 0.5);
    explicit bernoulli_distribution(const param_type& parm);
    void reset();
    // generating functions
    template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);
    // property functions
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
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.

26.4.8.2.2 Class template binomial_distribution

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

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

where \( t \) and \( p \) are the parameters of the distribution.

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

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

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

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

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

Requires: 0 \leq p \leq 1 and 0 \leq t.
26.4 Random number generation

Effects: Constructs a binomial_distribution object; \( t \) and \( p \) correspond to the respective parameters of the distribution.

\[
\text{IntType } t() \text{ const;}
\]

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

\[
\text{double } p() \text{ const;}
\]

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

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

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

\[
P(i | p) = p \cdot (1 - p)^i.
\]

where \( p \) is the parameter of the distribution.

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

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

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

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

`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.

26.4.8.2.4 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

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

where $k$ and $p$ are the parameters of the distribution.

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

    // constructor and reset functions
    explicit negative_binomial_distribution(IntType k = 01, double p = 0.5);
    explicit negative_binomial_distribution(const param_type& parm);
    void reset();

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

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

explicit negative_binomial_distribution(IntType k = 01, double p = 0.5);

Requires: $0 \leq k \leq 1$ and $0 \leq p \leq 1$.

Effects: Constructs a negative_binomial_distribution object; $k$ and $p$ correspond to the respective parameters of the distribution.
Returns: The value of the \( k \) parameter with which the object was constructed.

\[
\text{double } p() \text{ const;}
\]

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

26.4.8.3 Poisson distributions

26.4.8.3.1 Class template poisson_distribution

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

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

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

\[
\text{template <class IntType = int>}
\text{class poisson_distribution}
\text{
\{ public:
\text{// types}
\text{typedef IntType result_type;}
\text{typedef unspecified param_type;}
\text{
\text{// constructors and reset functions}
\text{explicit poisson_distribution(double mean = 1.0);}
\text{explicit poisson_distribution(const param_type& parm);}
\text{void reset();}
\text{
\text{// generating functions}
\text{template <class UniformRandomNumberGenerator>
\text{result_type operator() (UniformRandomNumberGenerator& urng);}
\text{template <class UniformRandomNumberGenerator>
\text{result_type operator() (UniformRandomNumberGenerator& urng, const param_type& parm);}
\text{
\text{// property functions}
\text{double mean() const;}
\text{param_type param() const;}
\text{void param(const param_type& parm);}
\text{result_type min() const;}
\text{result_type max() const;}
\text{\};}
\]

\text{explicit poisson_distribution(double mean = 0.51.0);}
\]

Requires: \( 0 < \text{mean} \).

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

\[
\text{double mean() const;}
\]
26.4.8.3.2 Class template exponential_distribution

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

$$p(x | \lambda) = \lambda e^{-\lambda x}.$$ where $\lambda$ is the parameter of the distribution.

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

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

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

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

explicit exponential_distribution(RealType lambda = 1.0);

Requires: $0 < \lambda$.

Effects: Constructs a `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 Random number generation

26.4.8.3.3 Class template gamma_distribution

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

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

where \( \alpha \) is the parameter of the distribution.

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

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

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

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

2

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

*Requires:* \( 0 < \alpha \) and \( 0 < \beta \).

3

*Effects:* Constructs a gamma_distribution object; \( \alpha \) and \( \beta \) corresponds to the parameters of the distribution.

4

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

5

*Returns:* The value of the \( \beta \) parameter with which the object was constructed.
A `weibull_distribution` random number distribution produces random numbers \( x \geq 0 \) distributed according to the probability density function

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

where \( a \) and \( b \) are the parameters of the distribution.

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

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

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

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

**Requires:** \( 0 < a \) and \( 0 < b \).

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

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

**Returns:** The value of the \( b \) parameter with which the object was constructed.
26.4 Random number generation

26.4.8.3.5 Class template extreme_value_distribution

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

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

where \( a \) and \( b \) are the parameters of the distribution.

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

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

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

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

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

*Requires:* \( 0 < b \).

*Effects:* Constructs an extreme_value_distribution object; \( a \) and \( b \) correspond to the respective parameters of the distribution.

RealType a() const;

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

---

6) 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.
Returns: The value of the b parameter with which the object was constructed.

26.4.8.4 Normal distributions

26.4.8.4.1 Class template normal_distribution

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

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

where the distribution parameters $\mu$ and $\sigma$, respectively, are also known as this distribution's mean and the standard deviation. are the parameters of the distribution.

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

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

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

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

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

Requires: $0 < \text{stddev}$.

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

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

Returns: The value of the \texttt{stddev} parameter with which the object was constructed.

26.4.8.4.2 Class template \texttt{lognormal\_distribution}

A \texttt{lognormal\_distribution} random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

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

where \( m \) and \( s \) are the parameters of the distribution.

\begin{verbatim}
template <class RealType = double>
class lognormal_distribution
{
    public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

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

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

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

\texttt{explicit lognormal\_distribution(RealType m = 0.0, RealType s = 1.0);}  

Requirements: \( 0 < s \).

Effects: Constructs a \texttt{lognormal\_distribution} object; \( m \) and \( s \) correspond to the respective parameters of the distribution.

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
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.

26.4.8.4.3 Class template chi_squared_distribution

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

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

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

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

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

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

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

explicit chi_squared_distribution(int n = 1);

Requires: \( 0 < n \).

Effects: Constructs a chi_squared_distribution object; \( n \) corresponds to the parameter of the distribution.

int n() const;
26.4 Random number generation

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

26.4.8.4.4 Class template cauchy_distribution

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

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

where \( a \) and \( b \) are the parameters of the distribution.

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

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

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

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

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

Requires: \( 0 < b \).

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

RealType a() const;

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

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

26.4.8.4.5 Class template fisher_f_distribution
[lib.rand.dist.norm.f]

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

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

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

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

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

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

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

explicit fisher_f_distribution(int m = 1, int n = 1);

Requires: \( 0 < m \) and \( 0 < n \).

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

int m() const;

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
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 integer $n$ is the parameter of the distribution (a positive integer).

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

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

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

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

explicit student_t_distribution(int n = 1);

Returns: 0 < n.

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

int n() const;

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
Returns: The value of the n parameter with which the object was constructed.

26.4.8.5 Sampling distributions

26.4.8.5.1 Class template discrete_distribution

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

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

where the \( n \) probabilities \( p_i \) are the parameters of the distribution.

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

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

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

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

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]
26.4 Random number generation

NUMERICS LIBRARY

Effects: Constructs a discrete_distribution object with probabilities

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

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 \( k = 0, \ldots, n-1 \).

26.4.8.5.2 Class template piecewise_constant_distribution

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

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

where the \( n \) probabilities \( \rho_i \) and the corresponding \( n+1 \) interval boundaries \( b_i \) are the parameters of the distribution. The \( n+1 \) distribution parameters \( b_i \) are also known as this distribution’s interval boundaries.

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

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

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
piecewise_constant_distribution();

Effects: Constructs a piecewise_constant_distribution object with \( n = 1, p_0 = 1, b_0 = 0, \) and \( b_1 = 1. \)

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
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 \( p(x) \) whose shape is determined when the distribution is constructed. \( x_{\text{min}} \) and \( x_{\text{max}} \) are parameters of the distribution.

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

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

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

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

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

general_pdf_distribution();

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 Func>
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 $x_{\text{min}} \leq x < x_{\text{max}}$, $p_\text{df}(x)$ shall return a value that is non-negative, non-NaN, and non-infinity; and

c) the following relations shall hold:

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

where $f$ is the mathematical function corresponding to the supplied $p_\text{df}$. [Note: This implies that the user-supplied $p_\text{df}$ need not be normalized. — end note]

Effects: Constructs a general pdf distribution object; $x_{\text{min}}$ and $x_{\text{max}}$ correspond to the respective parameters of the distribution and the corresponding probability density function is given by $p(x)p(x) = p_\text{df}(x)/z$,

where $x_{\text{min}} \leq x < x_{\text{max}}$.

Returns: The value of the $x_{\text{min}}$ parameter with which the object was constructed.

Returns: The value of the $x_{\text{max}}$ parameter with which the object was constructed.
Index

a()
  cauchy_distribution<> , 40
  extreme_value_distribution<> , 36
  uniform_int_distribution<> , 27
  uniform_real_distribution<> , 28
  weibull_distribution<> , 35
alpha()
  gamma_distribution<> , 34
b()
  cauchy_distribution<> , 41
  extreme_value_distribution<> , 37
  uniform_int_distribution<> , 27
  uniform_real_distribution<> , 28
  weibull_distribution<> , 35
base()
  discard_block_engine<> , 18
  shuffle_order_engine<> , 20
base1()
  xor_combine_engine<> , 22
base2()
  xor_combine_engine<> , 22
Bernoulli distributions, 28–32
bernoulli_distribution , 28
  constructor, 29
  discrete probability function, 28
  p() , 29
beta()
  gamma_distribution<> , 34
binomial_distribution<> , 29
  constructor, 29
  discrete probability function, 29
  p() , 30
  t() , 30
carry
  subtract_with_carry_01_engine<> , 15
  subtract_with_carry_engine<> , 14
cauhcy_distribution<> , 40
  a() , 40
  b() , 41
  constructor, 40
  probability density function, 40
chi_squared_distribution<> , 39
  constructor, 39
  n() , 39
  probability density function, 39
densities()
  piecewise_constant_distribution<> , 45
discard_block_engine<> , 17
  base() , 18
  constructor, 18
  generation algorithm, 17
  state, 17
  template parameters, 18
  textual representation, 18
  transition algorithm, 17
discrete probability function, 6
  bernoulli_distribution , 28
  binomial_distribution<> , 29
  discrete_distribution<> , 43
  geometric_distribution<> , 30
  negative_binomial_distribution<> , 31
  poisson_distribution<> , 32
  uniform_int_distribution<> , 26
discrete_distribution<>
  discrete probability function, 43
discrete_distribution<> , 43
  constructor, 43
  discrete_distribution<> , 43
probabilities(), 44
weights, 44
distribution, see random number distribution

generation algorithm
discard_block_engine<> , 17
linear_congruential_engine<> , 11
mersenne_twister_engine<> , 13
random number engine, 3
shuffle_order_engine<> , 19
subtract_with_carry_01_engine<> , 16
subtract_with_carry_engine<> , 14
xor_combine_engine<> , 20
geometric_distribution<> , 30
constructor, 30
discrete probability function, 30
p(), 31

textual representation

interval boundaries
piecewise_constant_distribution<> , 44
intervals()
piecewise_constant_distribution<> , 45

knuth_b, 23

lambda()

linear_congruential_engine<> , 11
creator, 12
generation algorithm, 11
state, 11
template parameters, 12
textual representation, 12
transition algorithm, 11

lognormal_distribution<> , 38
creator, 38
m(), 39
probability density function, 38
s(), 39

m()
fisher_f_distribution<> , 41
lognormal_distribution<> , 39

max
random_device, 24
xor_combine_engine<> , 22

mean()

normal_distribution<> , 38
poisson_distribution<> , 32
student_t_distribution<> , 42

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
mersenne_twister_engine<>, 12
  constructor, 14
  generation algorithm, 13
  state, 12
  template parameters, 13
  textual representation, 13
  transition algorithm, 12

min
  random_device, 24
minstd_rand, 22
minstd_rand0, 22
mt19937, 22

n()
  chi_squared_distribution<>, 39
  fisher_f_distribution<>, 42
negative_binomial_distribution<>, 31
  constructor, 31
  discrete probability function, 31
  p(), 32
  t(), 31

normal distributions, 37–43
  normal_distribution<>, 37
    constructor, 37
    mean(), 38
    probability density function, 37
    stddev(), 38

operator()()
  random_device, 25

p()
  bernoulli_distribution, 29
  binomial_distribution<>, 30
  geometric_distribution<>, 31
  negative_binomial_distribution<>, 32

parameters
  random number distribution, 6
  piecewise_constant_distribution<>, 44
    constructor, 45
    densities(), 45
    interval boundaries, 44
    intervals(), 45
    probability density function, 44
    weights, 45
  Poisson distributions, 32–37
  poisson_distribution<>, 32
    constructor, 32
    discrete probability function, 32
    mean(), 32
  probabilities()
    discrete_distribution<>, 44
  probability density function, 6
  cauchy_distribution<>, 40
  chi_squared_distribution<>, 39
  exponential_distribution<>, 33
  extreme_value_distribution<>, 36
  fisher_f_distribution<>, 41
  gamma_distribution<>, 34
  general_pdf_distribution<>, 46
  lognormal_distribution<>, 38
  normal_distribution<>, 37
  piecewise_constant_distribution<>, 44
  student_t_distribution<>, 42
  uniform_real_distribution<>, 27
  weibull_distribution<>, 35

<long>, 8–11
random number distribution
  bernoulli_distribution, 28
  binomial_distribution<>, 29
  chi_squared_distribution<>, 39
  discrete probability function, 6
  discrete_distribution<>, 43
  exponential_distribution<>, 33
  extreme_value_distribution<>, 36
  fisher_f_distribution<>, 41
  gamma_distribution<>, 34
  general_pdf_distribution<>, 46
  geometric_distribution<>, 30
  lognormal_distribution<>, 38
  negative_binomial_distribution<>, 31
  normal_distribution<>, 37
  parameters, 6
  piecewise_constant_distribution<>, 44
  poisson_distribution<>, 32
  probability density function, 6
  requirements, 6–8
  student_t_distribution<>, 42
  uniform_int_distribution<>, 26
  uniform_real_distribution<>, 27
random number distributions
Bernoulli, 28–32
normal, 37–43
Poisson, 32–37
sampling, 43–47
uniform, 26–28
random number engine
  generation algorithm, 3
  linear_congruential_engine<>, 11
  mersenne_twister_engine<>., 12
  requirements, 2–5
  state, 3
  subtract_with_carry_01_engine<>., 15
  subtract_with_carry_engine<>., 14
  successor state, 3
  transition algorithm, 3
random number engine adaptor
  discard_block_engine<>., 17
  requirements, 5–6
  shuffle_order_engine<>., 18
  xor_combine_engine<>., 20
random number generation, 1–47
random number generator, see uniform random number generator
random_device, 23
  constructor, 24
  entropy(), 24
  implementation leeway, 24
  max, 24
  min, 24
  operator()(), 25
ranlux3, 23
ranlux3_01, 23
ranlux4, 23
ranlux4_01, 23
ranlux64_base_01, 23
ranlux_base_01, 23
requirements
  random number distribution, 6–8
  random number engine, 2–5
  random number engine adaptor, 5–6
  uniform random number generator, 2
result_type
  entity characterization based on, 1
  xor_combine_engine<>, 21
s()
lognormal_distribution<>, 39
sampling distributions, 43–47
shuffle_order_engine<>, 18
  base(), 20
  constructor, 20
  generation algorithm, 19
  state, 19
  template parameters, 19
  textual representation, 19
  transition algorithm, 19
state
  discard_block_engine<>, 17
  linear_congruential_engine<>, 11
  mersenne_twister_engine<>, 12
  random number engine, 3
  shuffle_order_engine<>, 19
  subtract_with_carry_01_engine<>, 15
  subtract_with_carry_engine<>, 14
  xor_combine_engine<>, 20
stddev()
  normal_distribution<>, 38
student_t_distribution<>, 42
  constructor, 42
  mean(), 42
  probability density function, 42
  subtract_with_carry_01_engine<>, 15
  carry, 15
  constructor, 16
  generation algorithm, 16
  state, 15
  template parameters, 16
  textual representation, 16
  transition algorithm, 15
subtract_with_carry_engine<>, 14
  carry, 14
  constructor, 15
  generation algorithm, 14
  state, 14
  template parameters, 15
  textual representation, 15
  transition algorithm, 14
successor state
  random number engine, 3
t()
  binomial_distribution<>, 30

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)
negative_binomial_distribution<>: 31

textual representation
discard_block_engine<>: 18
shuffle_order_engine<>: 19
subtract_with_carry_01_engine<>: 16
subtract_with_carry_engine<>: 15
xor_combine_engine<>: 22

transition algorithm
discard_block_engine<>: 17
linear_congruential_engine<>: 11
mersenne_twister_engine<>: 12
random number engine: 3
shuffle_order_engine<>: 19
subtract_with_carry_01_engine<>: 15
subtract_with_carry_engine<>: 14
xor_combine_engine<>: 20

uniform distributions: 26–28
uniform random number generator
requirements: 2
uniform_int_distribution<>: 26
 a(): 27
 b(): 27
 constructor: 27
discrete probability function: 26
uniform_real_distribution<>: 27
 a(): 28
 b(): 28
 constructor: 28
probability density function: 27

weibull_distribution<>: 35
 a(): 35
 b(): 35
 constructor: 35
probability density function: 35
weibull_distribution<>: 35

weights
discrete_distribution<>: 44
piecewise_constant_distribution<>: 45

xmax()
 general_pdf_distribution<>: 47
xmin()
 general_pdf_distribution<>: 47
xor_combine_engine<>: 20

Random Number Generation in C++0X: A Comprehensive Proposal, version 2 (N2032)