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


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

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

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

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

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

## List of Tables

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>26 Numerics library</td>
<td>1</td>
</tr>
<tr>
<td>26.4 Random number generation</td>
<td>1</td>
</tr>
<tr>
<td>26.4.1 Requirements</td>
<td>1</td>
</tr>
<tr>
<td>26.4.1.1 General requirements</td>
<td>1</td>
</tr>
<tr>
<td>26.4.1.2 Uniform random number generator requirements</td>
<td>2</td>
</tr>
<tr>
<td>26.4.1.3 Random number engine requirements</td>
<td>2</td>
</tr>
<tr>
<td>26.4.1.4 Random number engine adaptor requirements</td>
<td>5</td>
</tr>
<tr>
<td>26.4.1.5 Random number distribution requirements</td>
<td>6</td>
</tr>
<tr>
<td>26.4.2 Header <code>&lt;random&gt;</code> synopsis</td>
<td>9</td>
</tr>
<tr>
<td>26.4.3 Random number engine class templates</td>
<td>11</td>
</tr>
<tr>
<td>26.4.3.1 Class template <code>linear_congruential_engine</code></td>
<td>12</td>
</tr>
<tr>
<td>26.4.3.2 Class template <code>mersenne_twister_engine</code></td>
<td>13</td>
</tr>
<tr>
<td>26.4.3.3 Class template <code>subtract_with_carry_engine</code></td>
<td>14</td>
</tr>
<tr>
<td>26.4.3.4 Class template <code>subtract_with_carry_01_engine</code> [TO BE DELETED]</td>
<td>16</td>
</tr>
<tr>
<td>26.4.4 Random number engine adaptor class templates</td>
<td>17</td>
</tr>
<tr>
<td>26.4.4.1 Class template <code>discard_block_engine</code></td>
<td>18</td>
</tr>
<tr>
<td>26.4.4.2 Class template <code>random_bits_engine</code></td>
<td>19</td>
</tr>
<tr>
<td>26.4.4.3 Class template <code>shuffle_order_engine</code></td>
<td>20</td>
</tr>
<tr>
<td>26.4.4.4 Class template <code>xor_combine_engine</code></td>
<td>21</td>
</tr>
<tr>
<td>26.4.5 Engines and engine adaptors with predefined parameters</td>
<td>23</td>
</tr>
<tr>
<td>26.4.6 Class <code>random_device</code></td>
<td>24</td>
</tr>
<tr>
<td>26.4.7 Utilities</td>
<td>25</td>
</tr>
<tr>
<td>26.4.7.1 Class <code>seed_seq</code></td>
<td>25</td>
</tr>
<tr>
<td>26.4.7.2 Function template <code>generate_canonical</code></td>
<td>27</td>
</tr>
<tr>
<td>26.4.8 Random number distribution class templates</td>
<td>29</td>
</tr>
<tr>
<td>26.4.8.1 Uniform distributions</td>
<td>29</td>
</tr>
<tr>
<td>26.4.8.1.1 Class template <code>uniform_int_distribution</code></td>
<td>29</td>
</tr>
<tr>
<td>26.4.8.1.2 Class template <code>uniform_real_distribution</code></td>
<td>30</td>
</tr>
<tr>
<td>26.4.8.2 Bernoulli distributions</td>
<td>31</td>
</tr>
<tr>
<td>26.4.8.2.1 Class <code>bernoulli_distribution</code></td>
<td>31</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>26.4.8.2.2 Class template</td>
<td>32</td>
</tr>
<tr>
<td>binomial_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.2.3 Class template</td>
<td>33</td>
</tr>
<tr>
<td>geometric_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.2.4 Class template</td>
<td>34</td>
</tr>
<tr>
<td>negative_binomial_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.3 Poisson distributions</td>
<td>35</td>
</tr>
<tr>
<td>26.4.8.3.1 Class template</td>
<td>35</td>
</tr>
<tr>
<td>poisson_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.3.2 Class template</td>
<td>35</td>
</tr>
<tr>
<td>exponential_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.3.3 Class template</td>
<td>36</td>
</tr>
<tr>
<td>gamma_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.3.4 Class template</td>
<td>37</td>
</tr>
<tr>
<td>weibull_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.3.5 Class template</td>
<td>38</td>
</tr>
<tr>
<td>extreme_value_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.4 Normal distributions</td>
<td>40</td>
</tr>
<tr>
<td>26.4.8.4.1 Class template</td>
<td>40</td>
</tr>
<tr>
<td>normal_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.4.2 Class template</td>
<td>41</td>
</tr>
<tr>
<td>lognormal_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.4.3 Class template</td>
<td>42</td>
</tr>
<tr>
<td>chi_squared_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.4.4 Class template</td>
<td>43</td>
</tr>
<tr>
<td>cauchy_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.4.5 Class template</td>
<td>44</td>
</tr>
<tr>
<td>fisher_f_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.4.6 Class template</td>
<td>45</td>
</tr>
<tr>
<td>student_t_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.5 Sampling distributions</td>
<td>46</td>
</tr>
<tr>
<td>26.4.8.5.1 Class template</td>
<td>46</td>
</tr>
<tr>
<td>discrete_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.5.2 Class template</td>
<td>47</td>
</tr>
<tr>
<td>piecewise_constant_distribution</td>
<td></td>
</tr>
<tr>
<td>26.4.8.5.3 Class template</td>
<td>48</td>
</tr>
<tr>
<td>general_pdf_distribution</td>
<td></td>
</tr>
</tbody>
</table>

Index

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Table Description</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>6</td>
</tr>
<tr>
<td>4</td>
<td>Random number distribution requirements</td>
<td>7</td>
</tr>
</tbody>
</table>
This subclause defines a facility for generating (pseudo-)random numbers.

In addition to a few utilities, four categories of entities are described: uniform random number generators, random number engines, random number engine adaptors, and random number distributions. These categorizations are applicable to types that satisfy the corresponding requirements, to objects instantiated from such types, and to templates producing such types when instantiated. [Note: These entities are specified in such a way as to permit the binding of any uniform random number generator object e as the argument to any random number distribution object d, thus producing a zero-argument function object such as given by bind(d,e).—end note]

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

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

b) otherwise as integral or equivalently as integer-valued, if 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.

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

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

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

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

26.4.1 Requirements

26.4.1.1 General requirements

Throughout this subclause 26.4, the effect of instantiating a template

a) that has a template type parameter named UniformRandomNumberGenerator is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of uniform random number generator [26.4.1.2].
b) that has a template type parameter named \texttt{Engine} is undefined unless the corresponding template argument is \texttt{cv-unqualified} and satisfies the requirements of uniform random number engine [26.4.1.3].

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

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

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

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

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

A class \texttt{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) \texttt{T} is the type named by \texttt{X}'s associated \texttt{result_type}, and

b) \texttt{u} is a value of \texttt{X}.

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

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

A class \texttt{X} that \texttt{X} satisfies the requirements of a uniform random number generator [26.4.1.2] also satisfies the requirements of a random number engine if the expressions shown in table 2 are valid and have the indicated semantics, and if \texttt{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 \texttt{integral\_arithmetic} type [basic.fundamental];
e) \( g \) is an lvalue, of a type other than \( X \), that defines a zero-argument function object returning values of type \texttt{unsigned long\_q} is an lvalue of type \texttt{seed\_seq} [26.4.7.1];
f) \( z \) is a value of type \texttt{size\_t};
g) \( \texttt{os} \) is an lvalue of the type of some class template specialization \texttt{basic\_ostream<\charT, traits>>}; and
h) \( \texttt{is} \) is an lvalue of the type of some class template specialization \texttt{basic\_istream<\charT, traits>>};
where \( \charT \) and \( \texttt{traits} \) are constrained according to [lib.strings] and [lib.input.output].

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

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 \textit{transition algorithm} \( TA \) by which the engine’s state \( x_i \) is advanced to its \textit{successor state} \( x_{i+1} \), and
b) the \textit{generation algorithm} \( GA \) by which an engine’s state is mapped to a value of type \texttt{result\_type}.

Table 2: Random number engine requirements

<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>( \mathcal{O}(\text{size of state}) )</td>
</tr>
<tr>
<td>( X(x) )</td>
<td>—</td>
<td>Creates an engine that compares equal to ( x ).</td>
<td>( \mathcal{O}(\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;\texttt{unsigned long}\texttt{X::result_type}&gt;(s)}.</td>
<td>( \mathcal{O}(\text{size of state}) )</td>
</tr>
</tbody>
</table>
### Expression | Return Type | Pre/Post-condition | Complexity |
---|---|---|---|
$X(gq)$\(^1\) | — | With $n = q\cdot size()$, $C$ creates an engine $u$ with initial state determined by the results of successive invocations of $g$ as follows: If $n$ is 0, $u == X();$ otherwise, the initial state depends on a sequence produced by one call to $q\cdot randomize$. Throws what and when $g$ throws. | $O(\max(n, size of state))$ |
$u.seed()$ | void | post: $u == X()$ | same as $X()$ |
$u.seed(s)$ | void | post: $u == X(s)$ | same as $X(s)$ |
$u.seed(gq)$ | void | post: If $g$ does not throw, $u == v$, where the state of $v$ is as if constructed by $X(gq)$. Otherwise, the exception is rethrown and the engine’s state is deemed invalid. Thereafter, further use of $u$ is undefined except for destruction or invoking a function that establishes a valid state. | same as $X(gq)$ |
$u()$ | $T$ | Sets the state to $u_{i+1} = TA(u_i)$ and returns $GA(u_i)$. | amortized constant |
$u.discard(z)$\(^2\) | void | post: The state of $u$ is identical to that produced by $z$ consecutive calls to $u()$. | no worse than the complexity of $z$ consecutive calls to $u()$ |
$x == y$ | bool | With $S_x$ and $S_y$ as the infinite sequences of values that would be generated by repeated future calls to $x()$ and $y()$, respectively, returns true if $S_x = S_y$; returns false otherwise. | $O\text{(size of state)}$ |
$x != y$ | bool | !(x == y) | $O\text{(size of state)}$ |

---

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

\(^2\)This operation is common in user code, and can often be implemented in an engine-specific manner so as to provide significant performance improvements over an equivalent naive loop that makes $z$ consecutive calls to $u()$. 

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>os &lt;&lt; x</code></td>
<td>reference to the type of <code>os</code></td>
<td>With <code>os.fmtflags</code> 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</code> [lib.iostate.flags]). 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.fmtflags</code> are unchanged.</td>
<td><code>O(size of state)</code></td>
</tr>
</tbody>
</table>

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

5 If `Gen` is an arithmetic type [basic.fundamental], constructors instantiated from template $<\text{class Gen}> X(\text{Gen}& g)$ as well as member functions instantiated from template $<\text{class Gen}> \text{void seed(\text{Gen}& g)}$ 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 seed that takes a single integer argument instead of the version that takes a reference to a function object. — end note]

6 If a textual representation written via `os << x` was subsequently read via `is >> v`, then `x == v` provided that there have been no intervening invocations of `x` or of `v`.

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

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,

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

b) $b_i$ 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>$X::\text{base}_i_\text{type}$</td>
<td>$B_i$</td>
<td>—</td>
<td>compile time</td>
</tr>
<tr>
<td>$X::\text{base}_i()$</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:

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

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

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

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

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

$X$ shall have one additional constructor with $n$ or more parameters such that the type of parameter $i$, $1 \leq i \leq n$, is 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.

### 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,
a) $T$ is the type named by $X$'s associated $\text{result\_type}$;

b) $P$ is the type named by $X$'s associated $\text{param\_type}$;

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

d) $\text{glb}$ and $\text{lub}$ are values of $T$ respectively corresponding to the greatest lower bound and the least upper bound on the values potentially returned by $u$'s $\text{operator()}$, as determined by the current values of $u$'s parameters;

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

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

g) $\text{os}$ is an lvalue of the type of some class template specialization $\text{basic\_ostream<charT, traits>}$;

h) $\text{is}$ is an lvalue of the type of some class template specialization $\text{basic\_istream<charT, traits>}$;

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

2 The specification of each random number distribution identifies an associated mathematical probability density function $p(z)$ or an associated discrete probability function $P(z_i)$. Such functions are typically expressed using certain externally-supplied quantities known as the parameters of the distribution. Such distribution parameters are identified in this context by writing, for example, $p(z|a,b)$ or $P(z_i|a,b)$, to name specific parameters, or by writing, for example, $p(z|\{p\})$ 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>$X::\text{param_type}$</td>
<td>$P$</td>
<td>—</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>$u.\text{reset()}$</td>
<td>void</td>
<td>Subsequent uses of $u$ do not depend on values produced by $e$ prior to invoking $\text{reset}$.</td>
<td>constant</td>
</tr>
<tr>
<td>$x.\text{param()}$</td>
<td>$P$</td>
<td>Returns a value $p$ such that $X(p).\text{param()} == p$.</td>
<td>no worse than the complexity of $X(p)$</td>
</tr>
<tr>
<td>$u.\text{param}(p)$</td>
<td>void</td>
<td>post: $u.\text{param()} == p$.</td>
<td>no worse than the complexity of $X(p)$</td>
</tr>
<tr>
<td>$u(e)$</td>
<td>$T$</td>
<td>With $p = u.\text{param()}$, 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_i</td>
</tr>
</tbody>
</table>
### 26.4 Random number generation

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>pre/post-condition</th>
<th>complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>u(e, p)</code></td>
<td>T</td>
<td>The sequence of numbers returned by successive invocations with the same objects ( e ) and ( p ) is randomly distributed according to the associated ( p(z \mid {p}) ) or ( P(z_i \mid {p}) ) function.</td>
<td>—</td>
</tr>
<tr>
<td><code>x.min()</code></td>
<td>T</td>
<td>Returns ( \text{glb} ).</td>
<td>constant</td>
</tr>
<tr>
<td><code>x.max()</code></td>
<td>T</td>
<td>Returns ( \text{lub} ).</td>
<td>constant</td>
</tr>
<tr>
<td><code>os &lt;&lt; x</code></td>
<td>reference to the type of <code>os</code></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>—</td>
</tr>
<tr>
<td><code>is &gt;&gt; u</code></td>
<td>reference to the type of <code>is</code></td>
<td>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 <code>ios::failure</code> [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 ( \text{charT} ) and ( \text{traits} ) were the same as those of ( is ). post: The ( is.fmtflags ) are unchanged.</td>
<td>—</td>
</tr>
</tbody>
</table>

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

4 The sequence of numbers produced by repeated invocations of \( x(e) \) shall be independent of any invocation of \( os << x \) or of any \texttt{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 \( X::\text{param\_type} \) is declared as a (nested) \texttt{class} or via a \texttt{typedef}. In this subclause 26.4, declarations of \( X::\text{param\_type} \) are in the form of \texttt{typedef}s only for convenience of exposition.

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

8 For each of the constructors of \( X \) taking arguments corresponding to parameters of the distribution, P shall have a

---

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
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 {
    // \[lib.rand.synopsis\] Class template linear_congruential_engine
    template <class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine;

    // \[lib.rand.synopsis\] Class template mersenne_twister_engine
    template <class UIntType, int size_t w, int size_t n, int size_t m, int size_t r,
              UIntType a, int size_t u, int size_t s,
              UIntType b, int size_t t, UIntType c, int size_t l>
    class mersenne_twister_engine;

    // \[lib.rand.synopsis\] Class template subtract_with_carry_engine
    template <class UIntType, int size_t w, int size_t s, int size_t r>
    class subtract_with_carry_engine;

    // \[lib.rand.synopsis\] Class template discard_block_engine
    template <class Engine, int size_t p, int size_t r>
    class discard_block_engine;

    // \[lib.rand.synopsis\] Class template random_bits_engine
    template <class Engine, size_t w, class UIntType>
    class random_bits_engine;

    // \[lib.rand.synopsis\] Class template shuffle_order_engine
    template <class Engine, int size_t k>
    class shuffle_order_engine;

    // \[lib.rand.synopsis\] Class template xor_combine_engine
    template <class Engine1, int size_t s1, class Engine2, int size_t s2=0u>
    class xor_combine_engine;

    // Engines and engine adaptors with predefined parameters
    typedef see below minstd_rand0;
    typedef see below minstd_rand;
    typedef see below mt19937;
    typedef see below ranlux_base_01ranlux24_base;
    typedef see below ranlux64_base_01ranlux48_base;
    typedef see below ranlux3;
```

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
typedef see below ranlux4;
typedef see below ranlux24;
typedef see below ranlux48;
typedef see below knuth_b;

// [26.4.6] Class random_device
class random_device;

// [26.4.7.1] Class seed_seq
class seed_seq;

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

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

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

// [26.4.8.2.1] Class bernoulli_distribution
class bernoulli_distribution;

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

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

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

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

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

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

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

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

// [26.4.8.4.2] Class template lognormal_distribution
template <class RealType = double>
    class lognormal_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 [lib.rand.eng]

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

2 Except as required by table 2, no function described in this section 26.4.3 throws an exception.
The class templates specified in this section satisfy the requirements of random number engine. Descriptions are provided here only for operations on the engines that are not described in those requirements or for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment operators, for the bits_of_randomness member, and for equality and inequality operators are not shown in the synopsis.

### Class template `linear_congruential_engine`

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

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

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

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

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

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 is `numeric_limits<UIntType>::max() + 1`. [Note: The result need not be representable as a value of type `UIntType` — end note] Otherwise, the following relations shall hold: $a < m$ and $c < m$.

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

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

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

```cpp
template <class Gen> explicit linear_congruential_engine(Gen& g, seed_seq& q);
```
Effects: Constructs a `linear_congruential_engine` object. With \( y = g() \mod m \), \( k = \left\lceil \log_2 m \right\rceil \) and \( a \) an array (or equivalent) of length \( k + 3 \), invokes `q.randomize(a + 0, a + k + 3)` and then computes \( S = \left( \sum_{j=0}^{k-1} a_{j+3} \cdot 2^{32j} \right) \mod m \). If \( c \mod m \) is 0 and \( yS \) is 0, sets the engine’s state to 1, else sets the engine’s state to \( yS \).

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

26.4.3.2 Class template `mersenne_twister_engine`  

A `mersenne_twister_engine` random number engine\(^3\) produces unsigned integer random numbers in the closed interval \([0, 2^w - 1]\). The state \( x_i \) of a `mersenne_twister_engine` object \( x \) is of size \( n \) and consists of a sequence \( X \) of \( n \) values of the type delivered by \( x \); all subscripts applied to \( X \) are to be taken modulo \( n \).

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

The state transition is performed as follows:

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

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

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 \( z_2 = z_1 \text{xor} ((z_1 \text{lshift} w) \text{bitand} b) \).

c) Let \( z_3 = z_2 \text{xor} ((z_2 \text{lshift} w) \text{bitand} c) \).

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

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

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

\( ^3 \) The name of this engine refers, in part, to a property of its period: For properly-selected values of the parameters, the period is closely related to a large Mersenne prime number.
26.4 Random number generation

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

```c
// constructors and seeding functions
explicit mersenne_twister_engine(unsigned long result_type value = default_seed);
```

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

```c
void seed(unsigned long result_type value = default_seed);
```

```c
template <class Gen>
void seed(Gen& g seed_seq& q);
```

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

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

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

```c
explicit mersenne_twister_engine(unsigned long result_type value = default_seed);
```

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

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

```c
Complexity: $O(n)$.
```

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

```c
Effects: Constructs a mersenne_twister_engine object. Given the values ... obtained by successive invocations of g, sets ... to ..., respectively. With $k = [w/32]$ and $a$ an array (or equivalent) of length $n \cdot k$, invokes q.randomize($a + 0$, $a + n \cdot k$) and then, iteratively for $i = -n$, ..., $-1$, sets $X_i$ to \(\sum_{j=0}^{k-1} a_{k(i+n)+j} \cdot 2^{32j}\) mod $2^w$. Finally, if each of the resulting $X_i$ is 0, changes $X_{-n}$ to $2^w - 1$.

```c
Complexity: Exactly $n$ invocations of g.
```

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

The transition algorithm is a modular linear function of the form \( T_A(x_i) = (a \cdot x_i) \mod b \). The state transition is performed as follows:

- Let \( Y = X_{i-s} - X_{i-r} - c \).
- Set \( X_i \) to \( y = Y \mod m \). Set \( c \) to 1 if \( Y < 0 \), otherwise set \( c \) to 0.

[Note: This algorithm corresponds to a modular linear function of the form \( T_A(x_i) = (a \cdot x_i) \mod b \), where \( b \) is of the form \( m^r - m^s + 1 \) and \( a = b - \frac{b-1}{m} \). -- end note]

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

```
template <class UIntType, IntType msize_t w, intsize_t s, intsize_t r>
class subtract_with_carry_engine
{
public:
    // types
    typedef UIntType result_type;

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

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

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

The template parameter \( \text{IntType} \) shall denote a signed integral type large enough to store values up to \( m \).

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

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

```
extric explicit subtract_with_carry_engine(unsigned long result_type value = default_seed);
```

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
Required behavior: First construct \( e \), a linear_congruential_engine object, as if by the following definition:

\[
\text{linear_congruential_engine<}\text{unsigned_long, result_type}
\]

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

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

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

Effects: Constructs a subtract_with_carry_engine object. With \( \ldots \), sets \( k = \lceil w/32 \rceil \) and \( a \) an array (or equivalent) of length \( r \cdot k \), invokes \( q \cdot \text{randomize(a+0, a+r \cdot k)} \) and then, iteratively for \( i = -r, \ldots, -1 \), sets \( X_i \) to \( \left( \sum_{j=0}^{k-1} a_k(i) + j \cdot 2^{32j} \right) \mod m \). If \( X_{-1} \) is then 0, sets \( c \) to 1; otherwise sets \( c \) to 0.

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

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

Delete the entirety of this subsection.

A subtract_with_carry_01_engine random number engine produces floating-point random numbers. The state \( x_i \) of a subtract_with_carry_01_engine object \( x \) is of size \( \mathcal{O}(r) \), and consists of a sequence \( X \) of \( r \) integer values \( 0 \leq X_i < 2^w \); 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 \( T_A(x_i) = (a \cdot x_i) \mod p \), where \( p \) is of the form \( 2^w - 2^s + 1 \) and \( a = p - \frac{p-1}{2^s} \). The state transition is performed as follows:

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

b) Set \( X_i \) to \( Y \mod 2^w \). Set \( c \) to 1 if \( Y \) is negative, otherwise set \( c \) to 0.

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

The generation algorithm is \( G_A(x_i) = T \cdot 2^{-w} + \varepsilon \) 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 \( \varepsilon \) is \( 2^{-w(r+2)} \). [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;

  ...
This class template implements a random number generator that uses a subtract-with-carry method for its internal state. The state consists of a sequence of integers \(X_0, \ldots, X_{n-1}\), where \(n = \left\lfloor (w + 31)/32 \right\rfloor\), and a carry value \(c\). The generator uses a linear congruential engine to generate the sequence of integers and then applies the subtract-with-carry method to the sequence to produce the random numbers.

### Constructors and Seeding Functions

#### Explicit Constructor

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

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

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

#### Template Constructor

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

Constructs a `subtract_with_carry_01_engine` object. With \(n\) as above, sets the values of \(X_{n-1}, \ldots, X_0\), in that order, as follows. To set \(X_k\), obtain values \(z_0, \ldots, z_{n-1}\) by successive invocations of \(g\) taken modulo \(2^{32}\), and set \(X_k\) to \(\sum_{j=0}^{n-1} z_j \cdot 2^{32j}\). If \(X_{n-1}\) is then 0, sets \(c\) to 1; otherwise sets \(c\) to 0.

### Generating Functions

#### Explicit Operation

```cpp
result_type operator()();
```

### Complexity

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

### Notes

1. The latter relation ensures that \(\varepsilon\), used above, is representable as a non-zero value of `result_type`.
2. This algorithm ensures that only integer numbers representable in 32 bits are written.
for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment operators, and for equality and inequality operators are not shown in the synopses.

26.4.4.1 Class template discard_block_engine

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

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

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

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

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

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

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

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

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
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.

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

Returns: A reference to \( e \).

### 26.4.4.2 Class template random_bits_engine

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

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

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

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

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

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

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

    // generating functions
}
```

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
26.4 Random number generation

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

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

private:
    base_type e; // exposition only
};

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

The textual representation consists of the textual representation of $e$.

26.4.4.3 Class template shuffle_order_engine

A shuffle_order_engine random number engine adaptor produces the same random numbers that are produced by some base engine $e$, but delivers them in a different sequence. The state $x_i$ of a shuffle_order_engine engine adaptor object $x$ consists of the state $e_i$ of its base engine $e$, an additional value $Y$ of the type delivered by $e$, and an additional sequence $V$ of $k$ values also of the type delivered by $e$. The size of the state is the size of $e$’s state plus $k + 1$.

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

a) Calculate an integer $j$ as $\left\lfloor \frac{k(Y-b_{\text{min}})}{b_{\text{max}}-b_{\text{min}}} + 1 \right\rfloor$, if $e$ is integer-valued, or as $\lfloor k \cdot Y \rfloor$, if $e$ is real-valued.

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

The generation algorithm yields the last value of $Y$ produced while advancing $e$’s state as described above.

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

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

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
// generating functions
result_type operator()();
void discard(size_t z);

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

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

The following relation shall hold: $1 \leq k$.

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

In addition to its behavior pursuant to section 26.4.1.4, 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 $e()$.

const base_type& base() const;

Returns: A reference to $e$.

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

An xor_combine_engine random number engine adaptor produces random numbers from two integer-valued base engines $e_1$ and $e_2$ by merging their left-shifted random values via bitwise exclusive-or. The state $x_i$ of an xor_combine_engine engine adaptor object $x$ consists of the states $e_1$ and $e_2$ of its base engines. The size of the state is the size of the state of $e_1$ plus the size of the state of $e_2$.

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

The generation algorithm is $GA(x_i) = (e_1() \text{\ lshift}\ w s_1) \text{\ xor} (e_2() \text{\ lshift}\ w s_2)$, where $w$ denotes the value of numeric_limits<result_type>::digits and $v_1$ and $v_2$, respectively, denote the values of $(e_1() - e_1.\text{min})$ and $(e_2() - e_2.\text{min})$.

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

    // parameter values and engine characteristics
    static const int size_t shift1 = s1;
    static const int size_t shift2 = s2;
}
26.4 Random number generation

```cpp
static const result_type min = 0;
static const result_type max = see below;

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

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

// property functions
const base1_type& base1() const;
const base2_type& base2() const;
private:
    base1_type e1;    // exposition only
    base2_type e2;    // exposition only
};
```

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

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 \( s_2 \) be non-zero. (If both \( s_1 \) and \( s_2 \) are non-zero then the low bits will always be 0.)

b) It is also advisable for the unshifted base engine \( e_2 \)'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 value \( s_1 \) to be no greater than that \( n \).

— end note]

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

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

a) \( m_1 = \text{min}(\text{Engine1}\::\:\text{max} - \text{Engine1}\::\:\text{min}, 2^w \mbox{\text{-}} s_1 \mbox{\text{-}} 1) \),
b) \( m_2 = \text{min}(\text{Engine2}\::\:\text{max} - \text{Engine2}\::\:\text{min}, 2^w \mbox{\text{-}} s_2 \mbox{\text{-}} 1) \),

\[ A = m_1 \text{\ bitwise-and } m_2, \]
\[ B = m_1 \text{\ bitwise-or } m_2, \] and
\[ C = \ldots, \text{\ s = s1 - s2}. \]
23 Numerics library 26.4 Random number generation

the value of the member max is \((\text{C} \text{bitor} B)\text{shift} \ (s1 - s2), M(m1, m2, s)\text{shift} \ w, s2\), where \(M(a, b, d)\) is defined as follows:

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

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

Otherwise, if \(a \cdot 2^d \geq b\), define \(M(a, b, d) = (k + 1) \cdot 2^p + M((t \text{ mod } 2^p)/2^d, u \text{ mod } 2^p, d)\).

Otherwise, define \(M(a, b, d) = (k + 1) \cdot 2^p + M((u \text{ mod } 2^p)/2^d, t \text{ mod } 2^p, d)\).

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

\[
\text{const base}_t\text{ype& base1() const;}
\]

Returns: A reference to e1.

\[
\text{const base}_t\text{ype& base2() const;}
\]

Returns: A reference to e2.

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

typedef linear_congruential_engine<\text{unsigned long uint_fast32_t}, 16807, 0, 2147483647> minstd_rand0;

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

typedef linear_congruential_engine<\text{unsigned long uint_fast32_t}, 48271, 0, 2147483647> minstd_rand;

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

typedef mersenne_twister_engine<\text{unsigned long uint_fast32_t}, 32, 624, 397, 31, 0x9908b0df, 11, 7, 0x9d2c5680, 15, 0xefc60000, 18> mt19937;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type mt19937 shall produce the value 412365995.

typedef subtract_with_carry_01_engine<float uint_fast32_t, 24, 10, 24> ranlux_base_0124_base;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type ranlux_base_0124_base shall produce the value 7937952.

typedef subtract_with_carry_01_engine<\text{double uint_fast64_t}, 48, 5, 12> ranlux64_base_0148_base;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type ranlux64_base_0148_base shall produce the value 19211381361395461839128582725.
26.4 Random number generation

type
def discard_block_engine<subtract_with_carry_engine<unsigned long, (1<<24), 10, 24>,
223, 24> 

ranlux3;

6

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

type
def discard_block_engine<subtract_with_carry_engine<unsigned long, (1<<24), 10, 24>,
389, 24> 

ranlux4;

7

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

typedef discard_block_engine<ranlux_base_0124_base, 223, 2423> 
ranlux3_0124;

8

Required behavior: The 10000th consecutive invocation of a default-constructed object of type ranlux3_0124 shall produce the value 59576209901578.

typedef discard_block_engine<ranlux_base_0148_base, 389, 2411> 
ranlux4_0148;

9

Required behavior: The 10000th consecutive invocation of a default-constructed object of type ranlux4_0148 shall produce the value 8587295249142670248501.

typedef shuffle_order_engine<minstd_rand0, 256> 
knuth_b;

10

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

26.4.6 Class random_device

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

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

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

  // generator characteristics
  static const result_type min = see below;
  static const result_type max = see below;
}
// constructors
explicit random_device(const string& token = implementation-defined);

// generating functions
result_type operator()();

// property functions
double entropy() const;

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

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

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

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

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

double entropy() const;

Returns: If the implementation employs a random number engine, returns 0.0. Otherwise, returns an entropy estimate5) for the random numbers returned by operator(), in the range min(0) to log2(max(0) + 1).

Throws: Nothing.

result_type operator()();

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

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

26.4.7 Utilities

26.4.7.1 Class seed_seq

An object of type seed_seq consumes a sequence of integer-valued data and produces a fixed number of unsigned integer values, 0 ≤ i < 212, based on the consumed data. [Note: Such an object provides a mechanism to avoid replication of streams of random variates. This can be useful in applications requiring large numbers of random number engines. — end note]

4) The parameter is intended to allow an implementation to differentiate between different sources of randomness.
5) If a device has n states whose respective probabilities are P_0, ..., P_{n-1}, the device entropy S is defined as S = -\sum_{i=0}^{n-1} P_i \cdot \log P_i.
2 In addition to the requirements set forth below, instances of seed_seq shall meet the requirements of CopyConstructible [lib.copyconstructible] and of Assignable [lib.container.requirements].

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

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

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

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

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

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

4 **Throws:** Nothing.

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

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

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

```cpp
    for( v.clear(); n > 0; n -= 32 )
        v.push_back(b mod 2^32), b /= 2^32;
```

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

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

8 **Effects:** With s = v.size() and n = end - begin, fills the supplied range [begin, end) according to the following
algorithm\(^6\) in which each operation is to be carried out modulo \(2^{32}\), each indexing operator applied to \(\text{begin}\) is to be taken modulo \(n\), each indexing operator applied to \(v\) is to be taken modulo \(s\), and \(T(x)\) is defined as \(x \oplus (x \text{rshift } 30)\):

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

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

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

\[
(\text{begin}[k] \oplus (1664525 \cdot T(\text{begin}[k-1]))) + v[k] + (k \text{ mod } s).
\]

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

\[
(\text{begin}[k] \oplus (1566083941 \cdot T(\text{begin}[k-1]))) - k.
\]

Throws: Nothing.

size_t size() const;

Returns: The number of 32-bit units the object can deliver, as if by returning the result of \(v.\text{size}\).

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

Requires: OutputIterator shall satisfy the requirements of an output iterator [lib.output.iterator] such that \(\text{iterator_traits<OutputIterator>::value_type}\) shall be assignable from \(\text{result_type}\).

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

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

### 26.4.7.2 Function template generate_canonical

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

\[\text{Note: Obtaining a value in this way can be a useful step in the process of transforming a value generated by a uniform random number generator into a value that can be delivered by a random number distribution. — end note}\]

---

\(^6\)This algorithm is due to Dr. Makoto Matsumoto, with an improvement (approved by Dr. Matsumoto) suggested by Dr. Charles Karney.
For purposes of this section let $\mathcal{Z}$ consist of all values $t_j$ of type \texttt{result\_type} such that \(\epsilon\) as the value of
\texttt{numeric\_limits\<result\_type>::epsilon()}.

a) If \texttt{result\_type} is a floating-point type [basic.fundamental], \(result\_type(0) < t_j < result\_type(1) \epsilon \leq t_j \leq 1 - \epsilon\).

b) If \texttt{result\_type} is a signed or unsigned integral type [basic.fundamental], \texttt{numeric\_limits<result\_type>::min()} \(\leq t_j \leq \text{numeric\_limits<result\_type>::max()}\).

\begin{verbatim}
template<class result_type, class UniformRandomNumberGenerator, size_t bits=1u>
result_type generate_canonical(UniformRandomNumberGenerator& g);
\end{verbatim}

\textit{Complexity:} Exactly \(\lfloor b/\log_2 R \rfloor\) invocations of \texttt{g}, where \(b\)\(^7\) is the lesser of \texttt{numeric\_limits<result\_type>::digits} and \texttt{bits} (but not less than one), and \(R\) is the value of \texttt{g.max - g.min + 1}.

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

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

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

c) Otherwise, \(|\mathcal{G}| > |\mathcal{Z}|\) must hold, and the set of distinct values that \texttt{g} can produce shall be partitioned in an unspecified manner into \(|\mathcal{Z}|\) 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 \(\mathcal{Z}\). The unique value corresponding to the subset containing \(x\) shall be returned as the result of the function call.

Invokes \texttt{g()} \(k\) times to obtain values \(g_0, \ldots, g_{k-1}\), respectively. Calculates a quantity
\[ S = \sum_{i=0}^{k-1} (g_i - g.\min) \cdot R^i \]
using arithmetic of type \texttt{result\_type}, if floating-point, otherwise using exact arithmetic.

\textit{Returns:} A value from \(\mathcal{Z}\), subject to the following required behavior. With \(\epsilon\) as above, and with \(M\) as the value of \(1 + \text{numeric\_limits<result\_type>::max()}\), either
\[ \frac{S \cdot (1 - 2\epsilon)}{R^k - 1} + \epsilon \cdot \text{if result\_type is real-valued, or} \left\lfloor \frac{M \cdot (2S + 1)}{2R^k} \right\rfloor, \text{otherwise.} \]

\textit{Throws:} What and when \texttt{g} throws.

\(^7\)\(b\) is introduced to avoid any attempt to produce more bits of randomness than can be held in the \texttt{result\_type}.

---

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
The classes and class templates specified in this section 26.4.8 satisfy all the requirements of random number distribution [26.4.1.5]. Descriptions are provided here only for operations on the distributions that are not described in those requirements or for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment operators, and for equality and inequality operators are not shown in the synopses.

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

The value of each probability density function \( p(z) \) and of each discrete probability function \( P(z_i) \) specified in this section is 0 everywhere outside its stated domain.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
2 \textit{Requires:} a \leq b.

3 \textit{Effects:} Constructs a \texttt{uniform\_real\_distribution} object; a and b correspond to the respective parameters of the distribution.

result\_type a() const;

4 \textit{Returns:} The value of the a parameter with which the object was constructed.

result\_type b() const;

5 \textit{Returns:} The value of the b parameter with which the object was constructed.

26.4.8.2 Bernoulli distributions

26.4.8.2.1 Class \texttt{bernoulli\_distribution}

A \texttt{bernoulli\_distribution} random number distribution produces bool values b distributed according to the discrete probability function

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

```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;
};

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
26.4 Random number generation

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

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

```cpp
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 \(0 \leq t\).

Effects: Constructs a binomial_distribution object; \(t\) and \(p\) correspond to the respective parameters of the distribution.
\begin{verbatim}
33  Numerics library

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

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

26.4.8.2.3 Class template geometric_distribution

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

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

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

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

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

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

explicit geometric_distribution(double p = 0.5);

2 Requires: \( 0 < p < 1 \).

3 Effects: Constructs a geometric_distribution object; \( p \) corresponds to the parameter of the distribution.

5 Returns: The value of the \( p \) parameter with which the object was constructed.
\end{verbatim}
A `negative_binomial_distribution` random number distribution produces random integers \( i \geq 0 \) distributed according to the discrete probability function

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

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

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

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

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

1. `explicit negative_binomial_distribution(IntType k = 1, double p = 0.5);`
2. `Requires: 0 < p \leq 1 \text{ and } 0 < k.`
3. `Effects: Constructs a `negative_binomial_distribution` object; \( k \) and \( p \) correspond to the respective parameters of the distribution.
4. `IntType k() const;`
5. `Returns: The value of the \( k \) parameter with which the object was constructed.
6. `double p() const;`
7. `Returns: The value of the \( p \) parameter with which the object was constructed.`

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
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

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

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

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

    // constructors and reset functions
    explicit poisson_distribution(double mean = 1.0);
    explicit poisson_distribution(const param_type& parm);
    void reset();

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

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

explicit poisson_distribution(double mean = 1.0);

2 Requires: 0 < mean.
3 Effects: Constructs a poisson_distribution object; mean corresponds to the parameter of the distribution.
4 Returns: The value of the mean parameter with which the object was constructed.

26.4.8.3.2 Class template exponential_distribution

An exponential_distribution random number distribution produces random numbers \( x > 0 \) distributed according
26.4 Random number generation

Numerics library

to the probability density function

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

template <class RealType = double>
class exponential_distribution
{
public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;
    // constructors and reset functions
    explicit exponential_distribution(RealType lambda = 1.0);
    explicit exponential_distribution(const param_type& parm);
    void reset();
    // generating functions
    template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);
    // property functions
    RealType lambda() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};

explicit exponential_distribution(RealType lambda = 1.0);

2 Requires: \(0 < \lambda\).

3 Effects: Constructs a \(\text{exponential\_distribution}\) object; \(\lambda\) corresponds to the parameter of the distribution.

RealType lambda() const;

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

26.4.8.3.3 Class template \texttt{gamma\_distribution} [lib.rand.dist.pois.gamma]

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

\[ p(x|\alpha,\beta) = \frac{e^{-x/\beta}}{\beta^\alpha \cdot \Gamma(\alpha)} \cdot x^{\alpha-1}. \]
template <class RealType = double>
class gamma_distribution
{
public:
  // types
  typedef RealType result_type;
  typedef unspecified param_type;
  // constructors and reset functions
  explicit gamma_distribution(RealType alpha = 1.0, RealType beta = 1.0);
  explicit gamma_distribution(const param_type& parm);
  void reset();

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

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

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

Requires: 0 < alpha and 0 < beta.
Effects: Constructs a gamma_distribution object; alpha and beta correspond to the parameters of the distribution.

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

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

26.4.8.3.4 Class template weibull_distribution

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

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
26.4 Random number generation

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

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

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

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

explicit weibull_distribution(RealType a = 1.0, RealType b = 1.0);

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

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

RealType a() const;

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

RealType b() const;

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

26.4.8.3.5 Class template extreme_value_distribution

An extreme_value_distribution random number distribution produces random numbers $x$ distributed according to

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
the probability density function\(^8\)

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

```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;
};
```

\(^8\) The distribution corresponding to this probability density function is also known (with a possible change of variable) as the Gumbel Type I, the log-Weibull, or the Fisher-Tippett Type I distribution.
26.4.8.4 Normal distributions

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

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

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

```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.

RealType mean() const;

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

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

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

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

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

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

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

**Returns:** The value of the \( \text{stddev} \) parameter with which the object was constructed.

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

**Effects:** Constructs a lognormal_distribution object; \( m \) and \( s \) correspond to the respective parameters of the distribution.

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

Returns: The value of the $a$ 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

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

where $n$ is a positive integer.

```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;

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

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

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.

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
A fisher_f_distribution random number distribution produces random numbers \( x \geq 0 \) distributed according to the probability density function

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

where \( m \) and \( n \) are positive integers.

```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;`

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

`int n() const;`

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

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

where \( n \) is a positive integer.

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

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

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

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

explicit student_t_distribution(int n = 1);

Requires: \( 0 < n \).

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

int n() const;

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

26.4.8.5 Sampling distributions

26.4.8.5.1 Class template discrete_distribution

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

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

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

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

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

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

discrete_distribution();

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

template <class InputIterator>
 discrete_distribution(InputIterator firstW, InputIterator lastW);

Requires:

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

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

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

Effects: Constructs a discrete_distribution object with probabilities

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

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 [lib.rand.dist.samp.pconst]

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

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

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

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

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

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

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
26.4 Random number generation Numerics library 48

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

`piecewise_constant_distribution();`

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

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

**Requires:**

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

b) If `firstB == lastB`,

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

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

Otherwise,

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

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

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

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

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

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

`vector<result_type> intervals() const;`

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

`vector<double> densities() const;`

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

26.4.8.5.3 Class template `general_pdf_distribution` [lib.rand.distsamp.genpdf]

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

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)
according to the probability density function
\[ p(x \mid x_{\text{min}}, x_{\text{max}}, \rho) = \rho(x), \text{ for } x_{\text{min}} \leq x < x_{\text{min}}. \]

```cpp
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;
};
```

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

```cpp
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 \( x_{\text{min}} \leq x < x_{\text{max}} \), \( pdf(x) \) shall return a value that is non-negative, non-NaN, and non-infinity; and
c) the following relations shall hold:

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

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

**Effects:** Constructs a `general_pdf_distribution` object; \( x_{\text{min}} \) and \( x_{\text{max}} \) correspond to the respective parameters of the distribution and the corresponding probability density function is given by \( \rho(x) = \frac{f(x)}{z} \).

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

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

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

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

a()  
cauchy_distribution<>, 43  
extreme_value_distribution<>, 39  
uniform_int_distribution<>, 30  
uniform_real_distribution<>, 31  
weibull_distribution<>, 38  

alpha()  
gamma_distribution<>, 37  

b()  
cauchy_distribution<>, 43  
extreme_value_distribution<>, 39  
uniform_int_distribution<>, 30  
uniform_real_distribution<>, 31  
weibull_distribution<>, 38  

Bernoulli distributions, 31–34  
bernoulli_distribution, 31  
constructor, 32  
discrete probability function, 31  
p(), 32  

beta()  
gamma_distribution<>, 37  

binomial_distribution<>, 32  
constructor, 32  
discrete probability function, 32  
p(), 33  
t(), 33  

carry  
subtract_with_carry_engine<>, 15  

cauchy_distribution<>, 43  
a(), 43  
b(), 43  
constructor, 43  
probability density function, 43  

chi_squared_distribution<>, 42  
constructor, 42  
n(), 42  
probability density function, 42  

constructor  
random number engine requirement, 3, 4  

densities()  
piecewise_constant_distribution<>, 48  

discard()  
random number engine requirement, 4  
discard_block_engine<>, 18  
constructor, 19  
generation algorithm, 18  
state, 18  
template parameters, 19  
textual representation, 19  
transition algorithm, 18  

discrete probability function, 7  
bernoulli_distribution, 31  
binomial_distribution<>, 32  
discrete_distribution<>, 46  
geometric_distribution<>, 33  
negative_binomial_distribution<>, 34  
poisson_distribution<>, 35  
uniform_int_distribution<>, 29  

discrete_distribution<>  
discrete probability function, 46  
discrete_distribution<>, 46  
constructor, 46  
discrete_distribution<>, 46  
probabilities<>, 47  
weights, 47  
distribution, see random number distribution  
engine, see random number engine
engine adaptor, see random number engine adaptor

engines with predefined parameters
   knuth_b, 24
   minstd_rand, 23
   minstd_rand0, 23
   mt19937, 23
   ranlux24, 24
   ranlux24_base, 23
   ranlux48, 24
   ranlux48_base, 23

entropy()
   random_device, 25

exponential_distribution<> , 35
   constructor, 36
   lambda(), 36
   probability density function, 36

extreme_value_distribution<> , 38
   a(), 39
   b(), 39
   constructor, 39
   probability density function, 39

fisher_f_distribution<> , 44
   constructor, 44
   m(), 44
   n(), 44
   probability density function, 44

gamma_distribution<> , 36
   alpha(), 37
   beta(), 37
   constructor, 37
   probability density function, 36

general_pdf_distribution<> , 48
   constructor, 49
   probability density function, 49
   xmax(), 50
   xmin(), 50

generate_canonical<> , 27, 28

generation algorithm
   discard_block_engine<>, 18
   linear_congruential_engine<>, 12
   mersenne_twister_engine<>, 13
   random number engine, 3
   random_bits_engine<>, 19
   shuffle_order_engine<>, 20
   subtract_with_carry_engine<>, 15
   xor_combine_engine<>, 21

geometric_distribution<> , 33
   constructor, 33
   discrete probability function, 33
   p(), 33

generate_canonical<>()
   seed_seq, 27

interval boundaries
   piecewise_constant_distribution<> , 47
   intervals()
   piecewise_constant_distribution<> , 48

knuth_b, 24

lambda()
   exponential_distribution<> , 36

linear_congruential_engine<>, 12
   constructor, 12, 13
   generation algorithm, 12
   state, 12
   template parameters, 12
   textual representation, 12
   transition algorithm, 12

lognormal_distribution<> , 41
   constructor, 41
   m(), 41
   probability density function, 41
   s(), 41

max
   random_device, 25
   uniform random number generator requirement, 2
   xor_combine_engine<>, 23

generate_canonical<>()
   seed_seq, 27, 28

mean
   normal_distribution<> , 40
   poisson_distribution<> , 35
   mean()
   normal_distribution<> , 40
   poisson_distribution<> , 35

student_t_distribution<> , 45
   mersenne_twister_engine<>, 13
constructor, 14
generation algorithm, 13
state, 13
template parameters, 14
textual representation, 14
transition algorithm, 13

min
random_device, 25
uniform random number generator requirement, 2
minstd_rand, 23
minstd_rand0, 23
mt19937, 23

n()
chi_squared_distribution<>, 42
fisher_f_distribution<>, 44
negative_binomial_distribution<> constructor, 34
discrete probability function, 34
p(), 34
t(), 34

normal distributions, 40–45
normal_distribution<> constructor, 40
mean, 40
mean(), 40
probability density function, 40
standard deviation, 40
stddev(), 40

operator
operator==() random number engine requirement, 4
operator()() random number engine requirement, 4
random_device, 25
uniform random number generator requirement, 2
operator==() random number engine requirement, 4
operator()() random number engine requirement, 5
operator<<() random number engine requirement, 5
operator>>()() random number engine requirement, 5

p()
bernoulli_distribution, 32
binomial_distribution<>, 33
geometric_distribution<>, 33
negative_binomial_distribution<>, 34

parameters
random number distribution, 7
piecewise_constant_distribution<> constructor, 48
densities(), 48
interval boundaries, 47
intervals(), 48
probability density function, 47
weights, 48

Poisson distributions, 35–39
poisson_distribution<> constructor, 35
discrete probability function, 35
mean, 35
mean(), 35
probabilities() discrete_distribution<> probabilities() probability density function, 7
cauchy_distribution<> constructor, 43
chi_squared_distribution<> constructor, 42
exponential_distribution<> constructor, 36
extreme_value_distribution<> constructor, 38
fisher_f_distribution<> constructor, 44
gamma_distribution<> constructor, 36
general_pdf_distribution<> constructor, 49
lognormal_distribution<> constructor, 41
normal_distribution<> constructor, 40
piecewise_constant_distribution<> constructor, 47
student_t_distribution<> constructor, 45
uniform_real_distribution<> constructor, 30
weibull_distribution<> constructor, 37

<random>, 9–11
random number distribution
bernoulli_distribution, 31
binomial_distribution<> constructor, 32
chi_squared_distribution<> constructor, 42
discrete probability function, 7
discrete_distribution<> constructor, 46
exponential_distribution<> constructor, 35
extreme_value_distribution<> constructor, 38
fisher_f_distribution<> constructor, 44
gamma_distribution<> constructor, 36
INDEX

general_pdf_distribution<>, 48
geometric_distribution<>, 33
lognormal_distribution<>, 41
negative_binomial_distribution<>, 34
normal_distribution<>, 40
parameters, 7
piecewise_constant_distribution<>, 47
poisson_distribution<>, 35
probability density function, 7
requirements, 6–9
student_t_distribution<>, 45
uniform_int_distribution<>, 29
uniform_real_distribution<>, 30

random number distributions
Bernoulli, 31–34
normal, 40–45
Poisson, 35–39
sampling, 46–50
uniform, 29–31

random number engine
generation algorithm, 3
linear_congruential_engine<>, 12
mersenne_twister_engine<>, 13
requirements, 2–5
state, 3
subtract_with_carry_engine<>, 14
successor state, 3
transition algorithm, 3
with predefined parameters, 23–24

random number engine adaptor
discard_block_engine<>, 18
random_bits_engine<>, 19
requirements, 5–6
shuffle_order_engine<>, 20
with predefined parameters, 23–24
xor_combine_engine<>, 21

random number generation, 1–50
random number generator, see uniform random number generator
discard_block_engine<>, 18
random_bits_engine<>, 19
requirements, 5–6
shuffle_order_engine<>, 20
with predefined parameters, 23–24
xor_combine_engine<>, 21
constructor, 25
entropy(), 25
implementation leeway, 24
max, 25
min, 25
operator()(), 25
randomize()
    seed_seq, 26
ranlux24, 24
ranlux24_base, 23
ranlux48, 24
ranlux48_base, 23

requirements
random number distribution, 6–9
random number engine, 2–5
random number engine adaptor, 5–6
uniform random number generator, 2

result_type
destination characterization based on, 1
uniform random number generator requirement, 2
xor_combine_engine<>, 22

s()
lognormal_distribution<>, 41

sampling distributions, 46–50
seed()
    random number engine requirement, 4
seed_seq
    constructor, 26
    get_seeds(), 27
    overview, 25
    randomize(), 26
    size(), 27
shuffle_order_engine<>, 20
    constructor, 21
    generation algorithm, 20
    state, 20
template parameters, 21
textual representation, 21
transition algorithm, 20
size()
    seed_seq, 27
standard deviation
    normal_distribution<>, 40
state
discard_block_engine<> , 18
linear_congruential_engine<> , 12
mersenne_twister_engine<> , 13
random number engine , 3
random_bits_engine<> , 19
shuffle_order_engine<> , 20
subtract_with_carry_engine<> , 15
xor_combine_engine<> , 21

stddev()

normal_distribution<> , 40
student_t_distribution<> , 45
creator , 45
mean() , 45
probability density function , 45
subtract_with_carry_engine<> , 14
carry , 15
creator , 15, 16
generation algorithm , 15
state , 15
template parameters , 15
textual representation , 15
transition algorithm , 15

successor state

random number engine , 3

t()  

binomial_distribution<> , 33
negative_binomial_distribution<> , 34
textual representation
discard_block_engine<> , 19
random_bits_engine<> , 20
shuffle_order_engine<> , 21
subtract_with_carry_engine<> , 15
xor_combine_engine<> , 23

transition algorithm
discard_block_engine<> , 18
linear_congruential_engine<> , 12
mersenne_twister_engine<> , 13
random number engine , 3
random_bits_engine<> , 19
shuffle_order_engine<> , 20
subtract_with_carry_engine<> , 15
xor_combine_engine<> , 21

uniform distributions , 29–31
uniform random number generator

Random Number Generation in C++0X: A Comprehensive Proposal, version 3 (N2079)