Boost.Random

Jens Maurer

Copyright © 2000-2005 Jens Maurer
Copyright © 2009, 2010 Steven Watanabe

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

Introduction 2
Tutorial 3
  Generating integers in a range 3
  Generating integers with different probabilities 3
  Generating a random password 4
Reference 5
  Concepts 5
  Generators 10
  Distributions 12
  Headers 15
Performance 207
History and Acknowledgements 214

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Introduction

Random numbers are useful in a variety of applications. The Boost Random Number Library (Boost.Random for short) provides a variety of generators and distributions to produce random numbers having useful properties, such as uniform distribution.

You should read the concepts documentation for an introduction and the definition of the basic concepts. For a quick start, it may be sufficient to have a look at random_demo.cpp.

For a very quick start, here's an example:

```
boost::random::mt19937 rng;         // produces randomness out of thin air
                                    // see pseudo-random number generators
boost::random::uniform_int_distribution<> six(1,6);
                                    // distribution that maps to 1..6
                                    // see random number distributions
int x = six(rng);                   // simulate rolling a die
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Tutorial

Generating integers in a range

For the source of this example see die.cpp. First we include the headers we need for `mt19937` and `uniform_int_distribution`.

```
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/uniform_int_distribution.hpp>
```

We use `mt19937` with the default seed as a source of randomness. The numbers produced will be the same every time the program is run. One common method to change this is to seed with the current time (`std::time(0)` defined in ctime).

```
boost::random::mt19937 gen;
```

Note

We are using a *global* generator object here. This is important because we don't want to create a new pseudo-random number generator at every call

Now we can define a function that simulates an ordinary six-sided die.

```
int roll_die() {
    ❶boost::random::uniform_int_distribution<> dist(1, 6);
    ❷return dist(gen);
}
```

❶ `mt19937` produces integers in the range [0, 2^{32}-1]. However, we want numbers in the range [1, 6]. The distribution `uniform_int_distribution` performs this transformation.

> **Warning**
>
> Contrary to common C++ usage `uniform_int_distribution` does not take a *half-open range*. Instead it takes a *closed range*. Given the parameters 1 and 6, `uniform_int_distribution` can produce any of the values 1, 2, 3, 4, 5, or 6.

❷ A distribution is a function object. We generate a random number by calling `dist` with the generator.

Generating integers with different probabilities

For the source of this example see weighted_die.cpp.

```
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/discrete_distribution.hpp>

boost::mt19937 gen;
```

This time, instead of a fair die, the probability of rolling a 1 is 50% (!). The other five faces are all equally likely.

`discrete_distribution` works nicely here by allowing us to assign weights to each of the possible outcomes.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Tip

If your compiler supports `std::initializer_list`, you can initialize `discrete_distribution` directly with the weights.

```
double probabilities[] = {
    0.5, 0.1, 0.1, 0.1, 0.1, 0.1
};
boost::random::discrete_distribution<> dist(probabilities);
```

Now define a function that simulates rolling this die.

```
int roll_weighted_die() {
    ❶return dist(gen) + 1;
}
```

❶ Add 1 to make sure that the result is in the range [1,6] instead of [0,5].

Generating a random password

For the source of this example see password.cpp.

This example demonstrates generating a random 8 character password.

```
#include <boost/random/random_device.hpp>
#include <boost/random/uniform_int_distribution.hpp>

int main() {
    ❶std::string chars(
        "abcdefghijklmnopqrstuvwxyz"
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        "1234567890"
        "!@#$%^&*()"
        "`~-_=+[{]}\\|;:'\",<.>/? ");
    ❷boost::random::random_device rng;
    ❸boost::random::uniform_int_distribution<> index_dist(0, chars.size() - 1);
    for(int i = 0; i < 8; ++i) {
        std::cout << chars[index_dist(rng)];
    }
    std::cout << std::endl;
}
```

❶ We first define the characters that we're going to allow. This is pretty much just the characters on a standard keyboard.
❷ We use `random_device` as a source of entropy, since we want passwords that are not predictable.
❸ Finally we select 8 random characters from the string and print them to cout.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Reference

Concepts

Introduction

Random numbers are required in a number of different problem domains, such as

- numerics (simulation, Monte-Carlo integration)
- games (non-deterministic enemy behavior)
- security (key generation)
- testing (random coverage in white-box tests)

The Boost Random Number Generator Library provides a framework for random number generators with well-defined properties so that the generators can be used in the demanding numerics and security domains. For a general introduction to random numbers in numerics, see

> "Numerical Recipes in C: The art of scientific computing", William H. Press, Saul A. Teukolsky, William A. Vetterling, Brian P. Flannery, 2nd ed., 1992, pp. 274-328

Depending on the requirements of the problem domain, different variations of random number generators are appropriate:

- non-deterministic random number generator
- pseudo-random number generator
- quasi-random number generator

All variations have some properties in common, the concepts (in the STL sense) is called UniformRandomNumberGenerator. This concept will be defined in a subsequent section.

The goals for this library are the following:

- allow easy integration of third-party random-number generators
- provide easy-to-use front-end classes which model popular distributions
- provide maximum efficiency

Uniform Random Number Generator

A uniform random number generator provides a sequence of random numbers uniformly distributed on a given range. The range can be compile-time fixed or available (only) after run-time construction of the object.

The *tight lower bound* of some (finite) set S is the (unique) member l in S, so that for all v in S, l <= v holds. Likewise, the *tight upper bound* of some (finite) set S is the (unique) member u in S, so that for all v in S, v <= u holds.

In the following table, X denotes a number generator class returning objects of type T, and v is a const value of X.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 1. UniformRandomNumberGenerator requirements

expression	return type	pre/post-condition
`X::result_type`	`T`	`std::numeric_limits<T>::is_specialized` is `true`, `T` is LessThanComparable
`u.operator()()`	`T`	-
`v.min()`	`T`	tight lower bound on the set of all values returned by `operator()`. The return value of this function shall not change during the lifetime of the object.
`v.max()`	`T`	if `std::numeric_limits<T>::is_integer`, tight upper bound on the set of all values returned by `operator()`, otherwise, the smallest representable number larger than the tight upper bound on the set of all values returned by `operator()`. In any case, the return value of this function shall not change during the lifetime of the object.

The member functions `min`, `max`, and `operator()` shall have amortized constant time complexity.

Note

For integer generators (i.e. integer `T`), the generated values `x` fulfill `min() <= x <= max()`, for non-integer generators (i.e. non-integer `T`), the generated values `x` fulfill `min() <= x < max()`.

Rationale: The range description with min and max serves two purposes. First, it allows scaling of the values to some canonical range, such as [0..1). Second, it describes the significant bits of the values, which may be relevant for further processing.

The range is a closed interval [min,max] for integers, because the underlying type may not be able to represent the half-open interval [min,max+1). It is a half-open interval [min, max) for non-integers, because this is much more practical for borderline cases of continuous distributions.

Note

The UniformRandomNumberGenerator concept does not require `operator()(long)` and thus it does not fulfill the `RandomNumberGenerator` (std:25.2.11 [lib.alg.random.shuffle]) requirements. Use the `random_number_generator` adapter for that.

Rationale: `operator()(long)` is not provided, because mapping the output of some generator with integer range to a different integer range is not trivial.

Non-deterministic Uniform Random Number Generator

A non-deterministic uniform random number generator is a UniformRandomNumberGenerator that is based on some stochastic process. Thus, it provides a sequence of truly-random numbers. Examples for such processes are nuclear decay, noise of a Zehner diode, tunneling of quantum particles, rolling a die, drawing from an urn, and tossing a coin. Depending on the environment, inter-arrival times of network packets or keyboard events may be close approximations of stochastic processes.

render X
XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

The class `random_device` is a model for a non-deterministic random number generator.

Note

This type of random-number generator is useful for security applications, where it is important to prevent an outside attacker from guessing the numbers and thus obtaining your encryption or authentication key. Thus, models of this concept should be cautious not to leak any information, to the extent possible by the environment. For example, it might be advisable to explicitly clear any temporary storage as soon as it is no longer needed.

Pseudo-Random Number Generator

A pseudo-random number generator is a UniformRandomNumberGenerator which provides a deterministic sequence of pseudo-random numbers, based on some algorithm and internal state. `Linear congruential` and `inversive congruential` generators are examples of such pseudo-random number generators. Often, these generators are very sensitive to their parameters. In order to prevent wrong implementations from being used, an external testsuite should check that the generated sequence and the validation value provided do indeed match.

Donald E. Knuth gives an extensive overview on pseudo-random number generation in his book "The Art of Computer Programming, Vol. 2, 3rd edition, Addison-Wesley, 1997". The descriptions for the specific generators contain additional references.

Note

Because the state of a pseudo-random number generator is necessarily finite, the sequence of numbers returned by the generator will loop eventually.

In addition to the UniformRandomNumberGenerator requirements, a pseudo-random number generator has some additional requirements. In the following table, `X` denotes a pseudo-random number generator class, `u` is a value of `X`, `i` is a value of integral type, `s` is a value of a type which models SeedSeq, and `j` a value of type `unsigned long long`.

Table 2. PseudoRandomNumberGenerator requirements

expression	return type	pre/post-condition
`X()`	-	creates a generator with a default seed.
`X(i)`	-	creates a generator seeding it with the integer `i`.
`X(s)`	-	creates a generator setting its initial state from the SeedSeq `s`.
`u.seed(...)`	`void`	sets the current state to be identical to the state that would be created by the corresponding constructor.
`u.discard(j)`	`void`	Advances the generator by `j` steps as if by `j` calls to `u()`.

Classes which model a pseudo-random number generator shall also model EqualityComparable, i.e. implement `operator==`. Two pseudo-random number generators are defined to be *equivalent* if they both return an identical sequence of numbers starting from a given state.

Classes which model a pseudo-random number generator shall also model the Streamable concept, i.e. implement `operator<<` and `operator>>`. `operator<<` writes all current state of the pseudo-random number generator to the given `ostream` so that `operator>>` can restore the state at a later time. The state shall be written in a platform-independent manner, but it is assumed that the `locales`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

used for writing and reading be the same. The pseudo-random number generator with the restored state and the original at the just-written state shall be equivalent.

Classes which model a pseudo-random number generator should also model the CopyConstructible and Assignable concepts. However, note that the sequences of the original and the copy are strongly correlated (in fact, they are identical), which may make them unsuitable for some problem domains. Thus, copying pseudo-random number generators is discouraged; they should always be passed by (non-const) reference.

The classes `rand48`, `minstd_rand`, and `mt19937` are models for a pseudo-random number generator.

Note

This type of random-number generator is useful for numerics, games and testing. The non-zero arguments constructor(s) and the `seed()` member function(s) allow for a user-provided state to be installed in the generator. This is useful for debugging Monte-Carlo algorithms and analyzing particular test scenarios. The Streamable concept allows to save/restore the state of the generator, for example to re-run a test suite at a later time.

Seed Sequence

A SeedSeq represents a sequence of values that can be used to set the initial state of a PseudoRandomNumberGenerator. `i` and `j` are RandomAccessIterators whose `value_type` is an unsigned integer type with at least 32 bits.

Table 3. SeedSeq requirements

expression	return type	pre/post-condition	complexity
`s.generate(i, j)`	void	stores 32-bit values to all the elements in the iterator range defined by `i` and `j`	O(j - i)

The class `seed_seq` and every UniformRandomNumberGenerator provided by the library are models of SeedSeq.

Random Distribution

A random distribution produces random numbers distributed according to some distribution, given uniformly distributed random values as input. In the following table, `X` denotes a random distribution class returning objects of type `T`, `u` is a value of `X`, `x` and `y` are (possibly const) values of `X`, `P` is the `param_type` of the distribution, `p` is a value of `P`, and `e` is an lvalue of an arbitrary type that meets the requirements of a UniformRandomNumberGenerator, returning values of type `U`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 4. Random distribution requirements (in addition to CopyConstructible, and Assignable)

expression	return type	pre/post-condition	complexity
`X::result_type`	`T`	-	compile-time
`X::param_type`	`P`	A type that stores the parameters of the distribution, but not any of the state used to generate random variates. `param_type` provides the same set of constructors and accessors as the distribution.	compile-time
`X(p)`	`X`	Initializes a distribution from its parameters	O(size of state)
`u.reset()`	`void`	subsequent uses of `u` do not depend on values produced by any engine prior to invoking `reset`.	constant
`u(e)`	`T`	the sequence of numbers returned by successive invocations with the same object `e` is randomly distributed with the probability density function of the distribution	amortized constant number of invocations of `e`
`u(e, p)`	`T`	Equivalent to X(p)(e), but may use a different (and presumably more efficient) implementation	amortized constant number of invocations of `e` + O(size of state)
`x.param()`	`P`	Returns the parameters of the distribution	O(size of state)
`x.param(p)`	void	Sets the parameters of the distribution	O(size of state)
`x.min()`	`T`	returns the minimum value of the distribution	constant
`x.max()`	`T`	returns the maximum value of the distribution	constant
`x == y`	`bool`	Indicates whether the two distributions will produce identical sequences of random variates if given equal generators	O(size of state)
`x != y`	`bool`	`!(x == y)`	O(size of state)

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

expression	**return type**	**pre/post-condition**	**complexity**
`os << x`	`std::ostream&`	writes a textual representation for the parameters and additional internal data of the distribution `x` to `os`. post: The `os.fmtflags` and fill character are unchanged.	O(size of state)
`is >> u`	`std::istream&`	restores the parameters and additional internal data of the distribution `u`. pre: `is` provides a textual representation that was previously written by `operator<<` post: The `is.fmtflags` are unchanged.	O(size of state)

Additional requirements: The sequence of numbers produced by repeated invocations of `x(e)` does not change whether or not `os << x` is invoked between any of the invocations `x(e)`. 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)` produce the same sequence of random numbers as would repeated invocations of `x(e)`.

Generators

This library provides several pseudo-random number generators. The quality of a pseudo random number generator crucially depends on both the algorithm and its parameters. This library implements the algorithms as class templates with template value parameters, hidden in `namespace boost::random`. Any particular choice of parameters is represented as the appropriately specializing `typedef` in `namespace boost`.

Pseudo-random number generators should not be constructed (initialized) frequently during program execution, for two reasons. First, initialization requires full initialization of the internal state of the generator. Thus, generators with a lot of internal state (see below) are costly to initialize. Second, initialization always requires some value used as a "seed" for the generated sequence. It is usually difficult to obtain several good seed values. For example, one method to obtain a seed is to determine the current time at the highest resolution available, e.g. microseconds or nanoseconds. When the pseudo-random number generator is initialized again with the then-current time as the seed, it is likely that this is at a near-constant (non-random) distance from the time given as the seed for first initialization. The distance could even be zero if the resolution of the clock is low, thus the generator re-iterates the same sequence of random numbers. For some applications, this is inappropriate.

Note that all pseudo-random number generators described below are CopyConstructible and Assignable. Copying or assigning a generator will copy all its internal state, so the original and the copy will generate the identical sequence of random numbers. Often, such behavior is not wanted. In particular, beware of the algorithms from the standard library such as `std::generate`. They take a functor argument by value, thereby invoking the copy constructor when called.

The following table gives an overview of some characteristics of the generators. The cycle length is a rough estimate of the quality of the generator; the approximate relative speed is a performance measure, higher numbers mean faster random number generation.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 5. generators

generator	length of cycle	approx. memory requirements	approx. speed compared to fastest	comment
`minstd_rand0`	2^{31}-2	`sizeof(int32_t)`	16%	-
`minstd_rand`	2^{31}-2	`sizeof(int32_t)`	16%	-
`rand48`	2^{48}-1	`sizeof(uint64_t)`	64%	-
`ecuyer1988`	approx. 2^{61}	`2*sizeof(int32_t)`	7%	-
`knuth_b`	?	`257*sizeof(uint32_t)`	12%	-
`kreutzer1986`	?	`98*sizeof(uint32_t)`	37%	-
`taus88`	~2^{88}	`3*sizeof(uint32_t)`	100%	-
`hellekalek1995`	2^{31}-1	`sizeof(int32_t)`	2%	good uniform distribution in several dimensions
`mt11213b`	2^{11213}-1	`352*sizeof(uint32_t)`	100%	good uniform distribution in up to 350 dimensions
`mt19937`	2^{19937}-1	`625*sizeof(uint32_t)`	93%	good uniform distribution in up to 623 dimensions
`mt19937_64`	2^{19937}-1	`312*sizeof(uint64_t)`	38%	good uniform distribution in up to 311 dimensions
`lagged_fibonacci607`	~2^{32000}	`607*sizeof(double)`	59%	-
`lagged_fibonacci1279`	~2^{67000}	`1279*sizeof(double)`	59%	-
`lagged_fibonacci2281`	~2^{120000}	`2281*sizeof(double)`	61%	-
`lagged_fibonacci3217`	~2^{170000}	`3217*sizeof(double)`	62%	-
`lagged_fibonacci4423`	~2^{230000}	`4423*sizeof(double)`	59%	-
`lagged_fibonacci9689`	~2^{510000}	`9689*sizeof(double)`	61%	-
`lagged_fibonacci19937`	~$2^{1050000}$	`19937*sizeof(double)`	59%	-
`lagged_fibonacci23209`	~$2^{1200000}$	`23209*sizeof(double)`	61%	-

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

generator	**length of cycle**	**approx. memory requirements**	**approx. speed compared to fastest**	**comment**
`lagged_fibonacci44497`	~$2^{2300000}$	`44497*sizeof(double)`	59%	-
`ranlux3`	~10^{171}	`24*sizeof(int)`	5%	-
`ranlux4`	~10^{171}	`24*sizeof(int)`	3%	-
`ranlux64_3`	~10^{171}	`24*sizeof(int64_t)`	5%	-
`ranlux64_4`	~10^{171}	`24*sizeof(int64_t)`	3%	-
`ranlux3_01`	~10^{171}	`24*sizeof(float)`	5%	-
`ranlux4_01`	~10^{171}	`24*sizeof(float)`	3%	-
`ranlux64_3_01`	~10^{171}	`24*sizeof(double)`	5%	-
`ranlux64_4_01`	~10^{171}	`24*sizeof(double)`	3%	-
`ranlux24`	~10^{171}	`24*sizeof(uint32_t)`	5%	-
`ranlux48`	~10^{171}	`12*sizeof(uint64_t)`	3%	-

As observable from the table, there is generally a quality/performance/memory trade-off to be decided upon when choosing a random-number generator. The multitude of generators provided in this library allows the application programmer to optimize the trade-off with regard to his application domain. Additionally, employing several fundamentally different random number generators for a given application of Monte Carlo simulation will improve the confidence in the results.

If the names of the generators don't ring any bell and you have no idea which generator to use, it is reasonable to employ `mt19937` for a start: It is fast and has acceptable quality.

Note

These random number generators are not intended for use in applications where non-deterministic random numbers are required. See `random_device` for a choice of (hopefully) non-deterministic random number generators.

Distributions

In addition to the random number generators, this library provides distribution functions which map one distribution (often a uniform distribution provided by some generator) to another.

Usually, there are several possible implementations of any given mapping. Often, there is a choice between using more space, more invocations of the underlying source of random numbers, or more time-consuming arithmetic such as trigonometric functions. This interface description does not mandate any specific implementation. However, implementations which cannot reach certain values of the specified distribution or otherwise do not converge statistically to it are not acceptable.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 6. Uniform Distributions

distribution	explanation	example
`uniform_smallint`	discrete uniform distribution on a small set of integers (much smaller than the range of the underlying generator)	drawing from an urn
`uniform_int_distribution`	discrete uniform distribution on a set of integers; the underlying generator may be called several times to gather enough randomness for the output	drawing from an urn
`uniform_01`	continuous uniform distribution on the range [0,1); important basis for other distributions	-
`uniform_real_distribution`	continuous uniform distribution on some range [min, max) of real numbers	for the range [0, 2pi): randomly dropping a stick and measuring its angle in radians (assuming the angle is uniformly distributed)

Table 7. Bernoulli Distributions

distribution	explanation	example
`bernoulli_distribution`	Bernoulli experiment: discrete boolean valued distribution with configurable probability	tossing a coin (p=0.5)
`binomial_distribution`	counts outcomes of repeated Bernoulli experiments	tossing a coin 20 times and counting how many front sides are shown
`geometric_distribution`	measures distance between outcomes of repeated Bernoulli experiments	throwing a die several times and counting the number of tries until a "6" appears for the first time
`negative_binomial_distribution`	Counts the number of failures of repeated Bernoulli experiments required to get some constant number of successes.	flipping a coin and counting the number of heads that show up before we get 3 tails

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 8. Poisson Distributions

distribution	explanation	example
`poisson_distribution`	poisson distribution	counting the number of alpha particles emitted by radioactive matter in a fixed period of time
`exponential_distribution`	exponential distribution	measuring the inter-arrival time of alpha particles emitted by radioactive matter
`gamma_distribution`	gamma distribution	-
`weibull_distribution`	weibull distribution	-
`extreme_value_distribution`	extreme value distribution	-
`beta_distribution`	beta distribution	-
`laplace_distribution`	laplace distribution	-

Table 9. Normal Distributions

distribution	explanation	example
`normal_distribution`	counts outcomes of (infinitely) repeated Bernoulli experiments	tossing a coin 10000 times and counting how many front sides are shown
`lognormal_distribution`	lognormal distribution (sometimes used in simulations)	measuring the job completion time of an assembly line worker
`chi_squared_distribution`	chi-squared distribution	-
`cauchy_distribution`	Cauchy distribution	-
`fisher_f_distribution`	Fisher F distribution	-
`student_t_distribution`	Student t distribution	-

Table 10. Sampling Distributions

distribution	explanation	example
`discrete_distribution`	discrete distribution with specific probabilities	rolling an unfair die
`piecewise_constant_distribution`	-	-
`piecewise_linear_distribution`	-	-

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 11. Miscellaneous Distributions

distribution	explanation	example
`triangle_distribution`	triangle distribution	-
`uniform_on_sphere`	uniform distribution on a unit sphere of arbitrary dimension	choosing a random point on Earth (assumed to be a sphere) where to spend the next vacations

Headers

Header <boost/random/additive_combine.hpp>

```
namespace boost {
  namespace random {
    template<typename MLCG1, typename MLCG2> class additive_combine_engine;
    typedef additive_combine_engine< linear_congruential_en↵
gine< uint32_t, 40014, 0, 2147483563 >, linear_congruential_en↵
gine< uint32_t, 40692, 0, 2147483399 >> ecuyer1988;
  }
}
```

Class template additive_combine_engine

boost::random::additive_combine_engine

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/additive_combine.hpp>

template<typename MLCG1, typename MLCG2>
class additive_combine_engine {
public:
  // types
  typedef MLCG1              first_base;
  typedef MLCG2              second_base;
  typedef MLCG1::result_type result_type;

  // construct/copy/destruct
  additive_combine_engine();
  explicit additive_combine_engine(result_type);
  template<typename SeedSeq> explicit additive_combine_engine(SeedSeq &);
  additive_combine_engine(typename MLCG1::result_type,
                          typename MLCG2::result_type);
  template<typename It> additive_combine_engine(It &, It);

  // public static functions
  static result_type min();
  static result_type max();

  // public member functions
  void seed();
  void seed(result_type);
  template<typename SeedSeq> void seed(SeedSeq &);
  void seed(typename MLCG1::result_type, typename MLCG2::result_type);
  template<typename It> void seed(It &, It);
  result_type operator()();
  template<typename Iter> void generate(Iter, Iter);
  void discard(boost::uintmax_t);

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const additive_combine_engine &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
               const additive_combine_engine &);
  friend bool operator==(const additive_combine_engine &,
                         const additive_combine_engine &);
  friend bool operator!=(const additive_combine_engine &,
                         const additive_combine_engine &);

  // public data members
  static const bool has_fixed_range;
};
```

Description

An instantiation of class template `additive_combine_engine` models a pseudo-random number generator . It combines two multiplicative linear_congruential_engine number generators, i.e. those with c = 0. It is described in

> "Efficient and Portable Combined Random Number Generators", Pierre L'Ecuyer, Communications of the ACM, Vol. 31, No. 6, June 1988, pp. 742-749, 774

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

The template parameters MLCG1 and MLCG2 shall denote two different linear_congruential_engine number generators, each with c = 0. Each invocation returns a random number X(n) := (MLCG1(n) - MLCG2(n)) mod (m1 - 1), where m1 denotes the modulus of MLCG1.

`additive_combine_engine` public construct/copy/destruct

1.
   ```
   additive_combine_engine();
   ```

   Constructs an `additive_combine_engine` using the default constructors of the two base generators.

2.
   ```
   explicit additive_combine_engine(result_type seed);
   ```

   Constructs an `additive_combine_engine`, using seed as the constructor argument for both base generators.

3.
   ```
   template<typename SeedSeq> explicit additive_combine_engine(SeedSeq & seq);
   ```

   Constructs an `additive_combine_engine`, using seq as the constructor argument for both base generators.

   > **Warning**
   >
   > The semantics of this function are liable to change. A `seed_seq` is designed to generate all the seeds in one shot, but this seeds the two base engines independantly and probably ends up giving the same sequence to both.

4.
   ```
   additive_combine_engine(typename MLCG1::result_type seed1, 
                           typename MLCG2::result_type seed2);
   ```

   Constructs an `additive_combine_engine`, using `seed1` and `seed2` as the constructor argument to the first and second base generators, respectively.

5.
   ```
   template<typename It> additive_combine_engine(It & first, It last);
   ```

   Contructs an `additive_combine_engine` with values from the range defined by the input iterators first and last. first will be modified to point to the element after the last one used.

   Throws: `std::invalid_argument` if the input range is too small.

   Exception Safety: Basic

`additive_combine_engine` public static functions

1.
   ```
   static result_type min();
   ```

   Returns the smallest value that the generator can produce

2.
   ```
   static result_type max();
   ```

   Returns the largest value that the generator can produce

`additive_combine_engine` public member functions

1.
   ```
   void seed();
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Seeds an `additive_combine_engine` using the default seeds of the two base generators.

2.
```
void seed(result_type seed);
```

Seeds an `additive_combine_engine`, using `seed` as the seed for both base generators.

3.
```
template<typename SeedSeq> void seed(SeedSeq & seq);
```

Seeds an `additive_combine_engine`, using `seq` to seed both base generators.

See the warning on the corresponding constructor.

4.
```
void seed(typename MLCG1::result_type seed1,
          typename MLCG2::result_type seed2);
```

Seeds an `additive_combine` generator, using `seed1` and `seed2` as the seeds to the first and second base generators, respectively.

5.
```
template<typename It> void seed(It & first, It last);
```

Seeds an `additive_combine_engine` with values from the range defined by the input iterators first and last. first will be modified to point to the element after the last one used.

Throws: `std::invalid_argument` if the input range is too small.

Exception Safety: Basic

6.
```
result_type operator()();
```

Returns the next value of the generator.

7.
```
template<typename Iter> void generate(Iter first, Iter last);
```

Fills a range with random values

8.
```
void discard(boost::uintmax_t z);
```

Advances the state of the generator by `z`.

`additive_combine_engine` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const additive_combine_engine & r);
```

Writes the state of an `additive_combine_engine` to a `std::ostream`. The textual representation of an `additive_combine_engine` is the textual representation of the first base generator followed by the textual representation of the second base generator.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const additive_combine_engine & r);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Reads the state of an `additive_combine_engine` from a `std::istream`.

3.
```
friend bool operator==(const additive_combine_engine & x, 
                       const additive_combine_engine & y);
```

Returns: true iff the two `additive_combine_engines` will produce the same sequence of values.

4.
```
friend bool operator!=(const additive_combine_engine & lhs, 
                       const additive_combine_engine & rhs);
```

Returns: true iff the two `additive_combine_engines` will produce different sequences of values.

Type definition ecuyer1988

ecuyer1988

Synopsis

```
// In header: <boost/random/additive_combine.hpp>


typedef additive_combine_engine< linear_congruential_engine< uint32_t, 40014, 0, 2147483563 >, lin↵
ear_congruential_engine< uint32_t, 40692, 0, 2147483399 >> ecuyer1988;
```

Description

The specialization ecuyer1988 was suggested in

> "Efficient and Portable Combined Random Number Generators", Pierre L'Ecuyer, Communications of the ACM, Vol. 31, No. 6, June 1988, pp. 742-749, 774

Header <boost/random/bernoulli_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class bernoulli_distribution;
  }
}
```

Class template bernoulli_distribution

boost::random::bernoulli_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/bernoulli_distribution.hpp>

template<typename RealType = double>
class bernoulli_distribution {
public:
  // types
  typedef int  input_type;
  typedef bool result_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef bernoulli_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 0.5);

    // public member functions
    RealType p() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit bernoulli_distribution(const RealType & = 0.5);
  explicit bernoulli_distribution(const param_type &);

  // public member functions
  RealType p() const;
  bool min() const;
  bool max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();
  template<typename Engine> bool operator()(Engine &) const;
  template<typename Engine>
    bool operator()(Engine &, const param_type &) const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const bernoulli_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
                  const bernoulli_distribution &);
  friend bool operator==(const bernoulli_distribution &,
                         const bernoulli_distribution &);
  friend bool operator!=(const bernoulli_distribution &,
                         const bernoulli_distribution &);
};
```

Description

Instantiations of class template bernoulli_distribution model a random distribution . Such a random distribution produces bool values distributed with probabilities P(true) = p and P(false) = 1-p. p is the parameter of the distribution.

`bernoulli_distribution` public construct/copy/destruct

1.
   ```
   explicit bernoulli_distribution(const RealType & p = 0.5);
   ```

   Constructs a `bernoulli_distribution` object. p is the parameter of the distribution.

   Requires: 0 <= p <= 1

2.
   ```
   explicit bernoulli_distribution(const param_type & param);
   ```

   Constructs `bernoulli_distribution` from its parameters

`bernoulli_distribution` public member functions

1.
   ```
   RealType p() const;
   ```

   Returns: The "p" parameter of the distribution.

2.
   ```
   bool min() const;
   ```

   Returns the smallest value that the distribution can produce.

3.
   ```
   bool max() const;
   ```

   Returns the largest value that the distribution can produce.

4.
   ```
   param_type param() const;
   ```

   Returns the parameters of the distribution.

5.
   ```
   void param(const param_type & param);
   ```

   Sets the parameters of the distribution.

6.
   ```
   void reset();
   ```

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

7.
   ```
   template<typename Engine> bool operator()(Engine & eng) const;
   ```

   Returns: a random variate distributed according to the `bernoulli_distribution` .

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

8.
```
template<typename Engine>
  bool operator()(Engine & eng, const param_type & param) const;
```

Returns: a random variate distributed according to the `bernoulli_distribution` with parameters specified by param.

`bernoulli_distribution` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const bernoulli_distribution & bd);
```

Writes the parameters of the distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const bernoulli_distribution & bd);
```

Reads the parameters of the distribution from a `std::istream`.

3.
```
friend bool operator==(const bernoulli_distribution & lhs,
                       const bernoulli_distribution & rhs);
```

Returns true iff the two distributions will produce identical sequences of values given equal generators.

4.
```
friend bool operator!=(const bernoulli_distribution & lhs,
                       const bernoulli_distribution & rhs);
```

Returns true iff the two distributions will produce different sequences of values given equal generators.

Class param_type

boost::random::bernoulli_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/bernoulli_distribution.hpp>



class param_type {
public:
  // types
  typedef bernoulli_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 0.5);

  // public member functions
  RealType p() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
```
explicit param_type(RealType p = 0.5);
```

Constructs the parameters of the distribution.

Requires: 0 <= p <= 1

`param_type` public member functions

1.
```
RealType p() const;
```

Returns the p parameter of the distribution.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

Writes the parameters to a std::ostream.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const param_type & param);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Reads the parameters from a std::istream.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are equal.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are different.

Header <boost/random/beta_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class beta_distribution;
  }
}
```

Class template beta_distribution

boost::random::beta_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/beta_distribution.hpp>

template<typename RealType = double>
class beta_distribution {
public:
  // types
  typedef RealType result_type;
  typedef RealType input_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef beta_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 1.0, RealType = 1.0);

    // public member functions
    RealType alpha() const;
    RealType beta() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit beta_distribution(RealType = 1.0, RealType = 1.0);
  explicit beta_distribution(const param_type &);

  // public member functions
  template<typename URNG> RealType operator()(URNG &) const;
  template<typename URNG>
    RealType operator()(URNG &, const param_type &) const;
  RealType alpha() const;
  RealType beta() const;
  RealType min() const;
  RealType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const beta_distribution &);
  template<typename CharT, typename Traits>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
               const beta_distribution &);
  friend bool operator==(const beta_distribution &, const beta_distribution &);
  friend bool operator!=(const beta_distribution &, const beta_distribution &);
};
```

Description

The beta distribution is a real-valued distribution which produces values in the range [0, 1]. It has two parameters, alpha and beta.

It has $p(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$.

`beta_distribution` public construct/copy/destruct

1.
   ```
   explicit beta_distribution(RealType alpha = 1.0, RealType beta = 1.0);
   ```

   Constructs an `beta_distribution` from its "alpha" and "beta" parameters.

   Requires: alpha > 0, beta > 0

2.
   ```
   explicit beta_distribution(const param_type & param);
   ```

   Constructs an `beta_distribution` from its parameters.

`beta_distribution` public member functions

1.
   ```
   template<typename URNG> RealType operator()(URNG & urng) const;
   ```

   Returns a random variate distributed according to the beta distribution.

2.
   ```
   template<typename URNG>
     RealType operator()(URNG & urng, const param_type & param) const;
   ```

   Returns a random variate distributed accordint to the beta distribution with parameters specified by `param`.

3.
   ```
   RealType alpha() const;
   ```

   Returns the "alpha" parameter of the distribution.

4.
   ```
   RealType beta() const;
   ```

   Returns the "beta" parameter of the distribution.

5.
   ```
   RealType min() const;
   ```

   Returns the smallest value that the distribution can produce.

6.
   ```
   RealType max() const;
   ```

   Returns the largest value that the distribution can produce.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

7.
```
param_type param() const;
```

Returns the parameters of the distribution.

8.
```
void param(const param_type & param);
```

Sets the parameters of the distribution.

9.
```
void reset();
```

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

`beta_distribution` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const beta_distribution & wd);
```

Writes an `beta_distribution` to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const beta_distribution & wd);
```

Reads an `beta_distribution` from a `std::istream`.

3.
```
friend bool operator==(const beta_distribution & lhs, 
                       const beta_distribution & rhs);
```

Returns true if the two instances of `beta_distribution` will return identical sequences of values given equal generators.

4.
```
friend bool operator!=(const beta_distribution & lhs, 
                       const beta_distribution & rhs);
```

Returns true if the two instances of `beta_distribution` will return different sequences of values given equal generators.

Class param_type

boost::random::beta_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/beta_distribution.hpp>



class param_type {
public:
  // types
  typedef beta_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 1.0, RealType = 1.0);

  // public member functions
  RealType alpha() const;
  RealType beta() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
```
explicit param_type(RealType alpha = 1.0, RealType beta = 1.0);
```

Constructs a `param_type` from the "alpha" and "beta" parameters of the distribution.

Requires: alpha > 0, beta > 0

`param_type` public member functions

1.
```
RealType alpha() const;
```

Returns the "alpha" parameter of the distribtuion.

2.
```
RealType beta() const;
```

Returns the "beta" parameter of the distribution.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

Writes a `param_type` to a `std::ostream`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const param_type & param);
```

   Reads a `param_type` from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

   Returns true if the two sets of parameters are the same.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

   Returns true if the two sets of parameters are the different.

Header <boost/random/binomial_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename IntType = int, typename RealType = double> 
      class binomial_distribution;
  }
}
```

Class template binomial_distribution

boost::random::binomial_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/binomial_distribution.hpp>

template<typename IntType = int, typename RealType = double> 
class binomial_distribution {
public:
  // types
  typedef IntType  result_type;
  typedef RealType input_type; 

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef binomial_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(IntType = 1, RealType = 0.5);

    // public member functions
    IntType t() const;
    RealType p() const;

    // friend functions
    template<typename CharT, typename Traits> 
      friend std::basic_ostream< CharT, Traits > & 
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits> 
      friend std::basic_istream< CharT, Traits > & 
      operator>>(std::basic_istream< CharT, Traits > &, param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit binomial_distribution(IntType = 1, RealType = 0.5);
  explicit binomial_distribution(const param_type &);

  // public member functions
  template<typename URNG> IntType operator()(URNG &) const;
  template<typename URNG> IntType operator()(URNG &, const param_type &) const;
  IntType t() const;
  RealType p() const;
  IntType min() const;
  IntType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();

  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, 
               const binomial_distribution &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    operator>>(std::basic_istream< CharT, Traits > &, binomial_distribution &);
  friend bool operator==(const binomial_distribution &,
                         const binomial_distribution &);
  friend bool operator!=(const binomial_distribution &,
                         const binomial_distribution &);
};
```

Description

The binomial distribution is an integer valued distribution with two parameters, `t` and `p`. The values of the distribution are within the range [0,t].

The distribution function is $P(k) = \binom{t}{k} p^k (1-p)^{t-k}$.

The algorithm used is the BTRD algorithm described in

> "The generation of binomial random variates", Wolfgang Hormann, Journal of Statistical Computation and Simulation, Volume 46, Issue 1 & 2 April 1993 , pages 101 - 110

`binomial_distribution` public construct/copy/destruct

1.
   ```
   explicit binomial_distribution(IntType t = 1, RealType p = 0.5);
   ```

   Construct a `binomial_distribution` object. `t` and `p` are the parameters of the distribution.

   Requires: t >=0 && 0 <= p <= 1

2.
   ```
   explicit binomial_distribution(const param_type & param);
   ```

   Construct an `binomial_distribution` object from the parameters.

`binomial_distribution` public member functions

1.
   ```
   template<typename URNG> IntType operator()(URNG & urng) const;
   ```

   Returns a random variate distributed according to the binomial distribution.

2.
   ```
   template<typename URNG> 
     IntType operator()(URNG & urng, const param_type & param) const;
   ```

   Returns a random variate distributed according to the binomial distribution with parameters specified by `param`.

3.
   ```
   IntType t() const;
   ```

   Returns the `t` parameter of the distribution.

4.
   ```
   RealType p() const;
   ```

   Returns the `p` parameter of the distribution.

5.
   ```
   IntType min() const;
   ```

   Returns the smallest value that the distribution can produce.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6.
```
IntType max() const;
```

Returns the largest value that the distribution can produce.

7.
```
param_type param() const;
```

Returns the parameters of the distribution.

8.
```
void param(const param_type & param);
```

Sets parameters of the distribution.

9.
```
void reset();
```

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

`binomial_distribution` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const binomial_distribution & bd);
```

Writes the parameters of the distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             binomial_distribution & bd);
```

Reads the parameters of the distribution from a `std::istream`.

3.
```
friend bool operator==(const binomial_distribution & lhs,
                       const binomial_distribution & rhs);
```

Returns true if the two distributions will produce the same sequence of values, given equal generators.

4.
```
friend bool operator!=(const binomial_distribution & lhs,
                       const binomial_distribution & rhs);
```

Returns true if the two distributions could produce different sequences of values, given equal generators.

Class param_type

boost::random::binomial_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/binomial_distribution.hpp>



class param_type {
public:
  // types
  typedef binomial_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(IntType = 1, RealType = 0.5);

  // public member functions
  IntType t() const;
  RealType p() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
```
explicit param_type(IntType t = 1, RealType p = 0.5);
```

Construct a `param_type` object. `t` and `p` are the parameters of the distribution.

Requires: t >=0 && 0 <= p <= 1

`param_type` public member functions

1.
```
IntType t() const;
```

Returns the `t` parameter of the distribution.

2.
```
RealType p() const;
```

Returns the `p` parameter of the distribution.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

Writes the parameters of the distribution to a `std::ostream`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is, param_type & param);
```

   Reads the parameters of the distribution from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

   Returns true if the parameters have the same values.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

   Returns true if the parameters have different values.

Header <boost/random/cauchy_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class cauchy_distribution;
  }
}
```

Class template cauchy_distribution

boost::random::cauchy_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/cauchy_distribution.hpp>

template<typename RealType = double>
class cauchy_distribution {
public:
  // types
  typedef RealType input_type;
  typedef RealType result_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef cauchy_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 0.0, RealType = 1.0);

    // public member functions
    RealType median() const;
    RealType sigma() const;
    RealType a() const;
    RealType b() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit cauchy_distribution(RealType = 0.0, RealType = 1.0);
  explicit cauchy_distribution(const param_type &);

  // public member functions
  RealType median() const;
  RealType sigma() const;
  RealType a() const;
  RealType b() const;
  RealType min() const;
  RealType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();
  template<typename Engine> result_type operator()(Engine &);
  template<typename Engine>
    result_type operator()(Engine &, const param_type &);

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const cauchy_distribution &);
  template<typename CharT, typename Traits>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
               const cauchy_distribution &);
  friend bool operator==(const cauchy_distribution &,
                         const cauchy_distribution &);
  friend bool operator!=(const cauchy_distribution &,
                         const cauchy_distribution &);
};
```

Description

The cauchy distribution is a continuous distribution with two parameters, median and sigma.

It has $p(x) = \frac{\sigma}{\pi(\sigma^2 + (x-m)^2)}$

`cauchy_distribution` public construct/copy/destruct

1.
   ```
   explicit cauchy_distribution(RealType median = 0.0, RealType sigma = 1.0);
   ```

   Constructs a `cauchy_distribution` with the paramters `median` and `sigma`.

2.
   ```
   explicit cauchy_distribution(const param_type & param);
   ```

   Constructs a `cauchy_distribution` from it's parameters.

`cauchy_distribution` public member functions

1.
   ```
   RealType median() const;
   ```

   Returns: the "median" parameter of the distribution

2.
   ```
   RealType sigma() const;
   ```

   Returns: the "sigma" parameter of the distribution

3.
   ```
   RealType a() const;
   ```

   Returns: the "median" parameter of the distribution

4.
   ```
   RealType b() const;
   ```

   Returns: the "sigma" parameter of the distribution

5.
   ```
   RealType min() const;
   ```

   Returns the smallest value that the distribution can produce.

6.
   ```
   RealType max() const;
   ```

   Returns the largest value that the distribution can produce.

7.
   ```
   param_type param() const;
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

8.
```
void param(const param_type & param);
```

9.
```
void reset();
```

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

10.
```
template<typename Engine> result_type operator()(Engine & eng);
```

Returns: A random variate distributed according to the cauchy distribution.

11.
```
template<typename Engine>
  result_type operator()(Engine & eng, const param_type & param);
```

Returns: A random variate distributed according to the cauchy distribution with parameters specified by param.

`cauchy_distribution` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const cauchy_distribution & cd);
```

Writes the distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const cauchy_distribution & cd);
```

Reads the distribution from a `std::istream`.

3.
```
friend bool operator==(const cauchy_distribution & lhs,
                       const cauchy_distribution & rhs);
```

Returns true if the two distributions will produce identical sequences of values, given equal generators.

4.
```
friend bool operator!=(const cauchy_distribution & lhs,
                       const cauchy_distribution & rhs);
```

Returns true if the two distributions may produce different sequences of values, given equal generators.

Class param_type

boost::random::cauchy_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/cauchy_distribution.hpp>



class param_type {
public:
  // types
  typedef cauchy_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 0.0, RealType = 1.0);

  // public member functions
  RealType median() const;
  RealType sigma() const;
  RealType a() const;
  RealType b() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
```
explicit param_type(RealType median = 0.0, RealType sigma = 1.0);
```

   Constructs the parameters of the cauchy distribution.

`param_type` public member functions

1.
```
RealType median() const;
```

   Returns the median of the distribution.

2.
```
RealType sigma() const;
```

   Returns the sigma parameter of the distribution.

3.
```
RealType a() const;
```

   Returns the median of the distribution.

4.
```
RealType b() const;
```

   Returns the sigma parameter of the distribution.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`param_type` friend functions

1.
   ```
   template<typename CharT, typename Traits> 
     friend std::basic_ostream< CharT, Traits > & 
     operator<<(std::basic_ostream< CharT, Traits > & os, 
                const param_type & param);
   ```

   Writes the parameters to a std::ostream.

2.
   ```
   template<typename CharT, typename Traits> 
     friend std::basic_istream< CharT, Traits > & 
     operator>>(std::basic_istream< CharT, Traits > & is, 
                const param_type & param);
   ```

   Reads the parameters from a std::istream.

3.
   ```
   friend bool operator==(const param_type & lhs, const param_type & rhs);
   ```

   Returns true if the two sets of parameters are equal.

4.
   ```
   friend bool operator!=(const param_type & lhs, const param_type & rhs);
   ```

   Returns true if the two sets of parameters are different.

Header <boost/random/chi_squared_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class chi_squared_distribution;
  }
}
```

Class template chi_squared_distribution

boost::random::chi_squared_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/chi_squared_distribution.hpp>

template<typename RealType = double>
class chi_squared_distribution {
public:
  // types
  typedef RealType result_type;
  typedef RealType input_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef chi_squared_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 1);

    // public member functions
    RealType n() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit chi_squared_distribution(RealType = 1);
  explicit chi_squared_distribution(const param_type &);

  // public member functions
  template<typename URNG> RealType operator()(URNG &);
  template<typename URNG>
    RealType operator()(URNG &, const param_type &) const;
  RealType n() const;
  RealType min() const;
  RealType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const chi_squared_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
                chi_squared_distribution &);
  friend bool operator==(const chi_squared_distribution &,
                         const chi_squared_distribution &);
  friend bool operator!=(const chi_squared_distribution &,
                         const chi_squared_distribution &);
};
```

Description

The chi squared distribution is a real valued distribution with one parameter, `n`. The distribution produces values > 0.

The distribution function is $P(x) = \frac{x^{(n/2)-1}e^{-x/2}}{\Gamma(n/2)2^{n/2}}$.

`chi_squared_distribution` public construct/copy/destruct

1.
   ```
   explicit chi_squared_distribution(RealType n = 1);
   ```

   Construct a `chi_squared_distribution` object. `n` is the parameter of the distribution.

   Requires: t >=0 && 0 <= p <= 1

2.
   ```
   explicit chi_squared_distribution(const param_type & param);
   ```

   Construct an `chi_squared_distribution` object from the parameters.

`chi_squared_distribution` public member functions

1.
   ```
   template<typename URNG> RealType operator()(URNG & urng);
   ```

   Returns a random variate distributed according to the chi squared distribution.

2.
   ```
   template<typename URNG> 
     RealType operator()(URNG & urng, const param_type & param) const;
   ```

   Returns a random variate distributed according to the chi squared distribution with parameters specified by `param`.

3.
   ```
   RealType n() const;
   ```

   Returns the `n` parameter of the distribution.

4.
   ```
   RealType min() const;
   ```

   Returns the smallest value that the distribution can produce.

5.
   ```
   RealType max() const;
   ```

   Returns the largest value that the distribution can produce.

6.
   ```
   param_type param() const;
   ```

   Returns the parameters of the distribution.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

7.

```
void param(const param_type & param);
```

Sets parameters of the distribution.

8.

```
void reset();
```

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

`chi_squared_distribution` friend functions

1.

```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const chi_squared_distribution & c2d);
```

Writes the parameters of the distribution to a `std::ostream`.

2.

```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             chi_squared_distribution & c2d);
```

Reads the parameters of the distribution from a `std::istream`.

3.

```
friend bool operator==(const chi_squared_distribution & lhs,
                       const chi_squared_distribution & rhs);
```

Returns true if the two distributions will produce the same sequence of values, given equal generators.

4.

```
friend bool operator!=(const chi_squared_distribution & lhs,
                       const chi_squared_distribution & rhs);
```

Returns true if the two distributions could produce different sequences of values, given equal generators.

Class param_type

boost::random::chi_squared_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/chi_squared_distribution.hpp>



class param_type {
public:
  // types
  typedef chi_squared_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 1);

  // public member functions
  RealType n() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
```
explicit param_type(RealType n = 1);
```

Construct a `param_type` object. `n` is the parameter of the distribution.

Requires: t >=0 && 0 <= p <= 1

`param_type` public member functions

1.
```
RealType n() const;
```

Returns the `n` parameter of the distribution.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

Writes the parameters of the distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is, param_type & param);
```

Reads the parameters of the distribution from a `std::istream`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

Returns true if the parameters have the same values.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the parameters have different values.

Header <boost/random/discard_block.hpp>

```
namespace boost {
  namespace random {
    template<typename UniformRandomNumberGenerator, std::size_t p, 
             std::size_t r> 
      class discard_block_engine;
  }
}
```

Class template discard_block_engine

boost::random::discard_block_engine

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/discard_block.hpp>

template<typename UniformRandomNumberGenerator, std::size_t p, std::size_t r>
class discard_block_engine {
public:
  // types
  typedef UniformRandomNumberGenerator base_type;
  typedef base_type::result_type       result_type;

  // construct/copy/destruct
  discard_block_engine();
  explicit discard_block_engine(const base_type &);
  explicit discard_block_engine(base_type &&);
  explicit discard_block_engine(seed_type);
  template<typename SeedSeq> explicit discard_block_engine(SeedSeq &);
  template<typename It> discard_block_engine(It &, It);

  // public member functions
  void seed();
  void seed(seed_type);
  template<typename SeedSeq> void seed(SeedSeq &);
  template<typename It> void seed(It &, It);
  const base_type & base() const;
  result_type operator()();
  void discard(boost::uintmax_t);
  template<typename It> void generate(It, It);

  // public static functions
  static result_type min();
  static result_type max();

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const discard_block_engine &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, discard_block_engine &);
  friend bool operator==(const discard_block_engine &,
                         const discard_block_engine &);
  friend bool operator!=(const discard_block_engine &,
                         const discard_block_engine &);

  // public data members
  static const std::size_t block_size;
  static const std::size_t used_block;
  static const bool has_fixed_range;
  static const std::size_t total_block;
  static const std::size_t returned_block;
};
```

Description

The class template discard_block_engine is a model of pseudo-random number generator . It modifies another generator by discarding parts of its output. Out of every block of `p` results, the first `r` will be returned and the rest discarded.

Requires: 0 < p <= r

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`discard_block_engine` public construct/copy/destruct

1.
   ```
   discard_block_engine();
   ```

   Uses the default seed for the base generator.

2.
   ```
   explicit discard_block_engine(const base_type & rng);
   ```

   Constructs a new `discard_block_engine` with a copy of rng.

3.
   ```
   explicit discard_block_engine(base_type && rng);
   ```

   Constructs a new `discard_block_engine` with rng.

4.
   ```
   explicit discard_block_engine(seed_type value);
   ```

   Creates a new `discard_block_engine` and seeds the underlying generator with `value`

5.
   ```
   template<typename SeedSeq> explicit discard_block_engine(SeedSeq & seq);
   ```

   Creates a new `discard_block_engine` and seeds the underlying generator with `seq`

6.
   ```
   template<typename It> discard_block_engine(It & first, It last);
   ```

   Creates a new `discard_block_engine` and seeds the underlying generator with first and last.

`discard_block_engine` public member functions

1.
   ```
   void seed();
   ```

   default seeds the underlying generator.

2.
   ```
   void seed(seed_type s);
   ```

   Seeds the underlying generator with s.

3.
   ```
   template<typename SeedSeq> void seed(SeedSeq & seq);
   ```

   Seeds the underlying generator with seq.

4.
   ```
   template<typename It> void seed(It & first, It last);
   ```

   Seeds the underlying generator with first and last.

5.
   ```
   const base_type & base() const;
   ```

   Returns the underlying engine.

6.
   ```
   result_type operator()();
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns the next value of the generator.

7.
```
void discard(boost::uintmax_t z);
```

8.
```
template<typename It> void generate(It first, It last);
```

`discard_block_engine` public static functions

1.
```
static result_type min();
```

Returns the smallest value that the generator can produce. This is the same as the minimum of the underlying generator.

2.
```
static result_type max();
```

Returns the largest value that the generator can produce. This is the same as the maximum of the underlying generator.

`discard_block_engine` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const discard_block_engine & s);
```

Writes a `discard_block_engine` to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             discard_block_engine & s);
```

Reads a `discard_block_engine` from a `std::istream`.

3.
```
friend bool operator==(const discard_block_engine & x, 
                       const discard_block_engine & y);
```

Returns true if the two generators will produce identical sequences.

4.
```
friend bool operator!=(const discard_block_engine & x, 
                       const discard_block_engine & y);
```

Returns true if the two generators will produce different sequences.

Header <boost/random/discrete_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename IntType = int, typename WeightType = double> 
      class discrete_distribution;
  }
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Class template discrete_distribution

boost::random::discrete_distribution

Synopsis

```
// In header: <boost/random/discrete_distribution.hpp>

template<typename IntType = int, typename WeightType = double> 
class discrete_distribution {
public:
  // types
  typedef WeightType input_type; 
  typedef IntType    result_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef discrete_distribution distribution_type;

    // construct/copy/destruct
    param_type();
    template<typename Iter> param_type(Iter, Iter);
    param_type(const std::initializer_list< WeightType > &);
    template<typename Range> explicit param_type(const Range &);
    template<typename Func> param_type(std::size_t, double, double, Func);

    // public member functions
    std::vector< WeightType > probabilities() const;

    // friend functions
    template<typename CharT, typename Traits> 
      friend std::basic_ostream< CharT, Traits > & 
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits> 
      friend std::basic_istream< CharT, Traits > & 
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  discrete_distribution();
  template<typename Iter> discrete_distribution(Iter, Iter);
  discrete_distribution(std::initializer_list< WeightType >);
  template<typename Range> explicit discrete_distribution(const Range &);
  template<typename Func> 
    discrete_distribution(std::size_t, double, double, Func);
  explicit discrete_distribution(const param_type &);

  // public member functions
  template<typename URNG> IntType operator()(URNG &) const;
  template<typename URNG> IntType operator()(URNG &, const param_type &) const;
  result_type min() const;
  result_type max() const;
  std::vector< WeightType > probabilities() const;
  param_type param() const;
  void param(const param_type &);
  void reset();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, 
               const discrete_distribution &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
    operator>>(std::basic_istream< CharT, Traits > &, 
               const discrete_distribution &);
  friend bool operator==(const discrete_distribution &, 
                         const discrete_distribution &);
  friend bool operator!=(const discrete_distribution &, 
                         const discrete_distribution &);
};
```

Description

The class `discrete_distribution` models a random distribution . It produces integers in the range [0, n) with the probability of producing each value is specified by the parameters of the distribution.

`discrete_distribution` public construct/copy/destruct

1.
```
discrete_distribution();
```

   Creates a new `discrete_distribution` object that has $p(0) = 1$ and $p(i|i > 0) = 0$.

2.
```
template<typename Iter> discrete_distribution(Iter first, Iter last);
```

   Constructs a `discrete_distribution` from an iterator range. If `first` == `last`, equivalent to the default constructor. Otherwise, the values of the range represent weights for the possible values of the distribution.

3.
```
discrete_distribution(std::initializer_list< WeightType > wl);
```

   Constructs a `discrete_distribution` from a `std::initializer_list`. If the `initializer_list` is empty, equivalent to the default constructor. Otherwise, the values of the `initializer_list` represent weights for the possible values of the distribution. For example, given the distribution

```
discrete_distribution<> dist{1, 4, 5};
```

   The probability of a 0 is 1/10, the probability of a 1 is 2/5, the probability of a 2 is 1/2, and no other values are possible.

4.
```
template<typename Range> explicit discrete_distribution(const Range & range);
```

   Constructs a `discrete_distribution` from a Boost.Range range. If the range is empty, equivalent to the default constructor. Otherwise, the values of the range represent weights for the possible values of the distribution.

5.
```
template<typename Func> 
  discrete_distribution(std::size_t nw, double xmin, double xmax, Func fw);
```

   Constructs a `discrete_distribution` that approximates a function. If nw is zero, equivalent to the default constructor. Otherwise, the range of the distribution is [0, nw), and the weights are found by calling fw with values evenly distributed between $\mathtt{xmin} + \delta/2$ and $\mathtt{xmax} - \delta/2$, where $\delta = (\mathtt{xmax} - \mathtt{xmin})/\mathtt{nw}$.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6.
```
explicit discrete_distribution(const param_type & param);
```

Constructs a `discrete_distribution` from its parameters.

`discrete_distribution` public member functions

1.
```
template<typename URNG> IntType operator()(URNG & urng) const;
```

Returns a value distributed according to the parameters of the `discrete_distribution`.

2.
```
template<typename URNG>
  IntType operator()(URNG & urng, const param_type & param) const;
```

Returns a value distributed according to the parameters specified by param.

3.
```
result_type min() const;
```

Returns the smallest value that the distribution can produce.

4.
```
result_type max() const;
```

Returns the largest value that the distribution can produce.

5.
```
std::vector< WeightType > probabilities() const;
```

Returns a vector containing the probabilities of each value of the distribution. For example, given

```
discrete_distribution<> dist = { 1, 4, 5 };
std::vector<double> p = dist.param();
```

the vector, p will contain {0.1, 0.4, 0.5}.

If `WeightType` is integral, then the weights will be returned unchanged.

6.
```
param_type param() const;
```

Returns the parameters of the distribution.

7.
```
void param(const param_type & param);
```

Sets the parameters of the distribution.

8.
```
void reset();
```

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

`discrete_distribution` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const discrete_distribution & dd);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Writes a distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const discrete_distribution & dd);
```

Reads a distribution from a `std::istream`

3.
```
friend bool operator==(const discrete_distribution & lhs,
                       const discrete_distribution & rhs);
```

Returns true if the two distributions will return the same sequence of values, when passed equal generators.

4.
```
friend bool operator!=(const discrete_distribution & lhs,
                       const discrete_distribution & rhs);
```

Returns true if the two distributions may return different sequences of values, when passed equal generators.

Class param_type

boost::random::discrete_distribution::param_type

Synopsis

```
// In header: <boost/random/discrete_distribution.hpp>



class param_type {
public:
  // types
  typedef discrete_distribution distribution_type;

  // construct/copy/destruct
  param_type();
  template<typename Iter> param_type(Iter, Iter);
  param_type(const std::initializer_list< WeightType > &);
  template<typename Range> explicit param_type(const Range &);
  template<typename Func> param_type(std::size_t, double, double, Func);

  // public member functions
  std::vector< WeightType > probabilities() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

`param_type` public construct/copy/destruct

1.
```
param_type();
```

Constructs a `param_type` object, representing a distribution with $p(0) = 1$ and $p(k|k > 0) = 0$.

2.
```
template<typename Iter> param_type(Iter first, Iter last);
```

If `first` == `last`, equivalent to the default constructor. Otherwise, the values of the range represent weights for the possible values of the distribution.

3.
```
param_type(const std::initializer_list< WeightType > & wl);
```

If wl.size() == 0, equivalent to the default constructor. Otherwise, the values of the `initializer_list` represent weights for the possible values of the distribution.

4.
```
template<typename Range> explicit param_type(const Range & range);
```

If the range is empty, equivalent to the default constructor. Otherwise, the elements of the range represent weights for the possible values of the distribution.

5.
```
template<typename Func>
  param_type(std::size_t nw, double xmin, double xmax, Func fw);
```

If nw is zero, equivalent to the default constructor. Otherwise, the range of the distribution is [0, nw), and the weights are found by calling fw with values evenly distributed between $\mathtt{xmin} + \delta/2$ and $\mathtt{xmax} - \delta/2$, where $\delta = (\mathtt{xmax} - \mathtt{xmin})/\mathtt{nw}$.

`param_type` public member functions

1.
```
std::vector< WeightType > probabilities() const;
```

Returns a vector containing the probabilities of each possible value of the distribution.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

Writes the parameters to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const param_type & param);
```

Reads the parameters from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns true if the two sets of parameters are the same.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are different.

Header <boost/random/exponential_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class exponential_distribution;
  }
}
```

Class template exponential_distribution

boost::random::exponential_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/exponential_distribution.hpp>

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

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef exponential_distribution distribution_type;

    // construct/copy/destruct
    param_type(RealType = 1.0);

    // public member functions
    RealType lambda() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit exponential_distribution(RealType = 1.0);
  explicit exponential_distribution(const param_type &);

  // public member functions
  RealType lambda() const;
  RealType min() const;
  RealType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();
  template<typename Engine> result_type operator()(Engine &) const;
  template<typename Engine>
    result_type operator()(Engine &, const param_type &) const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const exponential_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
               const exponential_distribution &);
  friend bool operator==(const exponential_distribution &,
                         const exponential_distribution &);
  friend bool operator!=(const exponential_distribution &,
                         const exponential_distribution &);
};
```

Description

The exponential distribution is a model of random distribution with a single parameter lambda.

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

`exponential_distribution` public construct/copy/destruct

1.
   ```
   explicit exponential_distribution(RealType lambda = 1.0);
   ```

   Constructs an `exponential_distribution` with a given lambda.

   Requires: lambda > 0

2.
   ```
   explicit exponential_distribution(const param_type & param);
   ```

   Constructs an `exponential_distribution` from its parameters

`exponential_distribution` public member functions

1.
   ```
   RealType lambda() const;
   ```

   Returns the lambda parameter of the distribution.

2.
   ```
   RealType min() const;
   ```

   Returns the smallest value that the distribution can produce.

3.
   ```
   RealType max() const;
   ```

   Returns the largest value that the distribution can produce.

4.
   ```
   param_type param() const;
   ```

   Returns the parameters of the distribution.

5.
   ```
   void param(const param_type & param);
   ```

   Sets the parameters of the distribution.

6.
   ```
   void reset();
   ```

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

7.
   ```
   template<typename Engine> result_type operator()(Engine & eng) const;
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns a random variate distributed according to the exponential distribution.

8.
```
template<typename Engine>
  result_type operator()(Engine & eng, const param_type & param) const;
```

Returns a random variate distributed according to the exponential distribution with parameters specified by param.

`exponential_distribution` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const exponential_distribution & ed);
```

Writes the distribution to a std::ostream.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const exponential_distribution & ed);
```

Reads the distribution from a std::istream.

3.
```
friend bool operator==(const exponential_distribution & lhs,
                       const exponential_distribution & rhs);
```

Returns true iff the two distributions will produce identical sequences of values given equal generators.

4.
```
friend bool operator!=(const exponential_distribution & lhs,
                       const exponential_distribution & rhs);
```

Returns true iff the two distributions will produce different sequences of values given equal generators.

Class param_type

boost::random::exponential_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/exponential_distribution.hpp>



class param_type {
public:
  // types
  typedef exponential_distribution distribution_type;

  // construct/copy/destruct
  param_type(RealType = 1.0);

  // public member functions
  RealType lambda() const;

  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
   ```
   param_type(RealType lambda = 1.0);
   ```

   Constructs parameters with a given lambda.

   Requires: lambda > 0

`param_type` public member functions

1.
   ```
   RealType lambda() const;
   ```

   Returns the lambda parameter of the distribution.

`param_type` friend functions

1.
   ```
   template<typename CharT, typename Traits> 
     friend std::basic_ostream< CharT, Traits > & 
     operator<<(std::basic_ostream< CharT, Traits > & os, 
                const param_type & param);
   ```

   Writes the parameters to a `std::ostream`.

2.
   ```
   template<typename CharT, typename Traits> 
     friend std::basic_istream< CharT, Traits > & 
     operator>>(std::basic_istream< CharT, Traits > & is, 
                const param_type & param);
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Reads the parameters from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are equal.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are different.

Header <boost/random/extreme_value_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class extreme_value_distribution;
  }
}
```

Class template extreme_value_distribution

boost::random::extreme_value_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/extreme_value_distribution.hpp>

template<typename RealType = double> 
class extreme_value_distribution {
public:
  // types
  typedef RealType result_type;
  typedef RealType input_type; 

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef extreme_value_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 1.0, RealType = 1.0);

    // public member functions
    RealType a() const;
    RealType b() const;

    // friend functions
    template<typename CharT, typename Traits> 
      friend std::basic_ostream< CharT, Traits > & 
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits> 
      friend std::basic_istream< CharT, Traits > & 
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit extreme_value_distribution(RealType = 1.0, RealType = 1.0);
  explicit extreme_value_distribution(const param_type &);

  // public member functions
  template<typename URNG> RealType operator()(URNG &) const;
  template<typename URNG> 
    RealType operator()(URNG &, const param_type &) const;
  RealType a() const;
  RealType b() const;
  RealType min() const;
  RealType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();

  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, 
               const extreme_value_distribution &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
    operator>>(std::basic_istream< CharT, Traits > &, 
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
               const extreme_value_distribution &);
  friend bool operator==(const extreme_value_distribution &,
                         const extreme_value_distribution &);
  friend bool operator!=(const extreme_value_distribution &,
                         const extreme_value_distribution &);
};
```

Description

The extreme value distribution is a real valued distribution with two parameters a and b.

It has $p(x) = \frac{1}{b}e^{\frac{a-x}{b} - e^{\frac{a-x}{b}}}$.

`extreme_value_distribution` public construct/copy/destruct

1.
```
explicit extreme_value_distribution(RealType a = 1.0, RealType b = 1.0);
```

   Constructs an `extreme_value_distribution` from its "a" and "b" parameters.

   Requires: b > 0

2.
```
explicit extreme_value_distribution(const param_type & param);
```

   Constructs an `extreme_value_distribution` from its parameters.

`extreme_value_distribution` public member functions

1.
```
template<typename URNG> RealType operator()(URNG & urng) const;
```

   Returns a random variate distributed according to the `extreme_value_distribution`.

2.
```
template<typename URNG>
  RealType operator()(URNG & urng, const param_type & param) const;
```

   Returns a random variate distributed accordint to the extreme value distribution with parameters specified by `param`.

3.
```
RealType a() const;
```

   Returns the "a" parameter of the distribution.

4.
```
RealType b() const;
```

   Returns the "b" parameter of the distribution.

5.
```
RealType min() const;
```

   Returns the smallest value that the distribution can produce.

6.
```
RealType max() const;
```

   Returns the largest value that the distribution can produce.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

7.
```
param_type param() const;
```

Returns the parameters of the distribution.

8.
```
void param(const param_type & param);
```

Sets the parameters of the distribution.

9.
```
void reset();
```

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

`extreme_value_distribution` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const extreme_value_distribution & wd);
```

Writes an `extreme_value_distribution` to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const extreme_value_distribution & wd);
```

Reads an `extreme_value_distribution` from a `std::istream`.

3.
```
friend bool operator==(const extreme_value_distribution & lhs, 
                       const extreme_value_distribution & rhs);
```

Returns true if the two instances of `extreme_value_distribution` will return identical sequences of values given equal generators.

4.
```
friend bool operator!=(const extreme_value_distribution & lhs, 
                       const extreme_value_distribution & rhs);
```

Returns true if the two instances of `extreme_value_distribution` will return different sequences of values given equal generators.

Class param_type

boost::random::extreme_value_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/extreme_value_distribution.hpp>



class param_type {
public:
  // types
  typedef extreme_value_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 1.0, RealType = 1.0);

  // public member functions
  RealType a() const;
  RealType b() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
```
explicit param_type(RealType a = 1.0, RealType b = 1.0);
```

Constructs a `param_type` from the "a" and "b" parameters of the distribution.

Requires: b > 0

`param_type` public member functions

1.
```
RealType a() const;
```

Returns the "a" parameter of the distribtuion.

2.
```
RealType b() const;
```

Returns the "b" parameter of the distribution.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

Writes a `param_type` to a `std::ostream`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const param_type & param);
```

Reads a `param_type` from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are the same.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are the different.

Header <boost/random/fisher_f_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class fisher_f_distribution;
  }
}
```

Class template fisher_f_distribution

boost::random::fisher_f_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/fisher_f_distribution.hpp>

template<typename RealType = double> 
class fisher_f_distribution {
public:
  // types
  typedef RealType result_type;
  typedef RealType input_type; 

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef fisher_f_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 1.0, RealType = 1.0);

    // public member functions
    RealType m() const;
    RealType n() const;

    // friend functions
    template<typename CharT, typename Traits> 
      friend std::basic_ostream< CharT, Traits > & 
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits> 
      friend std::basic_istream< CharT, Traits > & 
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit fisher_f_distribution(RealType = 1.0, RealType = 1.0);
  explicit fisher_f_distribution(const param_type &);

  // public member functions
  template<typename URNG> RealType operator()(URNG &);
  template<typename URNG> 
    RealType operator()(URNG &, const param_type &) const;
  RealType m() const;
  RealType n() const;
  RealType min() const;
  RealType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();

  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, 
               const fisher_f_distribution &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
    operator>>(std::basic_istream< CharT, Traits > &,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
               const fisher_f_distribution &);
  friend bool operator==(const fisher_f_distribution &,
                         const fisher_f_distribution &);
  friend bool operator!=(const fisher_f_distribution &,
                         const fisher_f_distribution &);
};
```

Description

The Fisher F distribution is a real valued distribution with two parameters m and n.

It has $p(x) = \frac{\Gamma((m+n)/2)}{\Gamma(m/2)\Gamma(n/2)} \left(\frac{m}{n}\right)^{m/2} x^{(m/2)-1} \left(1+\frac{mx}{n}\right)^{-(m+n)/2}$.

`fisher_f_distribution` public construct/copy/destruct

1.
```
explicit fisher_f_distribution(RealType m = 1.0, RealType n = 1.0);
```

   Constructs a `fisher_f_distribution` from its "m" and "n" parameters.

   Requires: m > 0 and n > 0

2.
```
explicit fisher_f_distribution(const param_type & param);
```

   Constructs an `fisher_f_distribution` from its parameters.

`fisher_f_distribution` public member functions

1.
```
template<typename URNG> RealType operator()(URNG & urng);
```

   Returns a random variate distributed according to the F distribution.

2.
```
template<typename URNG>
  RealType operator()(URNG & urng, const param_type & param) const;
```

   Returns a random variate distributed according to the F distribution with parameters specified by `param`.

3.
```
RealType m() const;
```

   Returns the "m" parameter of the distribution.

4.
```
RealType n() const;
```

   Returns the "n" parameter of the distribution.

5.
```
RealType min() const;
```

   Returns the smallest value that the distribution can produce.

6.
```
RealType max() const;
```

   Returns the largest value that the distribution can produce.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

7.
```
param_type param() const;
```

Returns the parameters of the distribution.

8.
```
void param(const param_type & param);
```

Sets the parameters of the distribution.

9.
```
void reset();
```

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

`fisher_f_distribution` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const fisher_f_distribution & fd);
```

Writes an `fisher_f_distribution` to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const fisher_f_distribution & fd);
```

Reads an `fisher_f_distribution` from a `std::istream`.

3.
```
friend bool operator==(const fisher_f_distribution & lhs, 
                       const fisher_f_distribution & rhs);
```

Returns true if the two instances of `fisher_f_distribution` will return identical sequences of values given equal generators.

4.
```
friend bool operator!=(const fisher_f_distribution & lhs, 
                       const fisher_f_distribution & rhs);
```

Returns true if the two instances of `fisher_f_distribution` will return different sequences of values given equal generators.

Class param_type

boost::random::fisher_f_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/fisher_f_distribution.hpp>



class param_type {
public:
  // types
  typedef fisher_f_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 1.0, RealType = 1.0);

  // public member functions
  RealType m() const;
  RealType n() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
   ```
   explicit param_type(RealType m = 1.0, RealType n = 1.0);
   ```

   Constructs a `param_type` from the "m" and "n" parameters of the distribution.

   Requires: m > 0 and n > 0

`param_type` public member functions

1.
   ```
   RealType m() const;
   ```

   Returns the "m" parameter of the distribtuion.

2.
   ```
   RealType n() const;
   ```

   Returns the "n" parameter of the distribution.

`param_type` friend functions

1.
   ```
   template<typename CharT, typename Traits>
     friend std::basic_ostream< CharT, Traits > &
     operator<<(std::basic_ostream< CharT, Traits > & os,
                const param_type & param);
   ```

   Writes a `param_type` to a `std::ostream`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
   ```
   template<typename CharT, typename Traits> 
     friend std::basic_istream< CharT, Traits > & 
     operator>>(std::basic_istream< CharT, Traits > & is, 
                const param_type & param);
   ```

   Reads a `param_type` from a `std::istream`.

3.
   ```
   friend bool operator==(const param_type & lhs, const param_type & rhs);
   ```

   Returns true if the two sets of parameters are the same.

4.
   ```
   friend bool operator!=(const param_type & lhs, const param_type & rhs);
   ```

   Returns true if the two sets of parameters are the different.

Header <boost/random/gamma_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class gamma_distribution;
  }
}
```

Class template gamma_distribution

boost::random::gamma_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/gamma_distribution.hpp>

template<typename RealType = double>
class gamma_distribution {
public:
  // types
  typedef RealType input_type;
  typedef RealType result_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef gamma_distribution distribution_type;

    // construct/copy/destruct
    param_type(const RealType & = 1.0, const RealType & = 1.0);

    // public member functions
    RealType alpha() const;
    RealType beta() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit gamma_distribution(const result_type & = 1.0,
                              const result_type & = 1.0);
  explicit gamma_distribution(const param_type &);

  // public member functions
  RealType alpha() const;
  RealType beta() const;
  RealType min() const;
  RealType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();
  template<typename Engine> result_type operator()(Engine &);
  template<typename URNG>
    RealType operator()(URNG &, const param_type &) const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const gamma_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    operator>>(std::basic_istream< CharT, Traits > &, gamma_distribution &);
  friend bool operator==(const gamma_distribution &, 
                         const gamma_distribution &);
  friend bool operator!=(const gamma_distribution &, 
                         const gamma_distribution &);
};
```

Description

The gamma distribution is a continuous distribution with two parameters alpha and beta. It produces values > 0.

It has $p(x) = x^{\alpha-1}\frac{e^{-x/\beta}}{\beta^\alpha\Gamma(\alpha)}$.

`gamma_distribution` public construct/copy/destruct

1.
```
explicit gamma_distribution(const result_type & alpha = 1.0, 
                            const result_type & beta = 1.0);
```

   Creates a new `gamma_distribution` with parameters "alpha" and "beta".

   Requires: alpha > 0 && beta > 0

2.
```
explicit gamma_distribution(const param_type & param);
```

   Constructs a `gamma_distribution` from its parameters.

`gamma_distribution` public member functions

1.
```
RealType alpha() const;
```

   Returns the "alpha" paramter of the distribution.

2.
```
RealType beta() const;
```

   Returns the "beta" parameter of the distribution.

3.
```
RealType min() const;
```

   Returns the smallest value that the distribution can produce.

4.
```
RealType max() const;
```

5.
```
param_type param() const;
```

   Returns the parameters of the distribution.

6.
```
void param(const param_type & param);
```

   Sets the parameters of the distribution.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

7.

```
void reset();
```

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

8.

```
template<typename Engine> result_type operator()(Engine & eng);
```

Returns a random variate distributed according to the gamma distribution.

9.

```
template<typename URNG>
  RealType operator()(URNG & urng, const param_type & param) const;
```

`gamma_distribution` friend functions

1.

```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const gamma_distribution & gd);
```

Writes a `gamma_distribution` to a `std::ostream`.

2.

```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             gamma_distribution & gd);
```

Reads a `gamma_distribution` from a `std::istream`.

3.

```
friend bool operator==(const gamma_distribution & lhs,
                       const gamma_distribution & rhs);
```

Returns true if the two distributions will produce identical sequences of random variates given equal generators.

4.

```
friend bool operator!=(const gamma_distribution & lhs,
                       const gamma_distribution & rhs);
```

Returns true if the two distributions can produce different sequences of random variates, given equal generators.

Class param_type

boost::random::gamma_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/gamma_distribution.hpp>



class param_type {
public:
  // types
  typedef gamma_distribution distribution_type;

  // construct/copy/destruct
  param_type(const RealType & = 1.0, const RealType & = 1.0);

  // public member functions
  RealType alpha() const;
  RealType beta() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
```
param_type(const RealType & alpha = 1.0, const RealType & beta = 1.0);
```

Constructs a `param_type` object from the "alpha" and "beta" parameters.

Requires: alpha > 0 && beta > 0

`param_type` public member functions

1.
```
RealType alpha() const;
```

Returns the "alpha" parameter of the distribution.

2.
```
RealType beta() const;
```

Returns the "beta" parameter of the distribution.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

Writes the parameters to a `std::ostream`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, param_type & param);
```

Reads the parameters from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are the same.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets fo parameters are different.

Header <boost/random/geometric_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename IntType = int, typename RealType = double> 
      class geometric_distribution;
  }
}
```

Class template geometric_distribution

boost::random::geometric_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/geometric_distribution.hpp>

template<typename IntType = int, typename RealType = double>
class geometric_distribution {
public:
  // types
  typedef RealType input_type;
  typedef IntType  result_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef geometric_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 0.5);

    // public member functions
    RealType p() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit geometric_distribution(const RealType & = 0.5);
  explicit geometric_distribution(const param_type &);

  // public member functions
  RealType p() const;
  IntType min() const;
  IntType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();
  template<typename Engine> result_type operator()(Engine &) const;
  template<typename Engine>
    result_type operator()(Engine &, const param_type &) const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const geometric_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
                 const geometric_distribution &);
  friend bool operator==(const geometric_distribution &,
                         const geometric_distribution &);
  friend bool operator!=(const geometric_distribution &,
                         const geometric_distribution &);
};
```

Description

An instantiation of the class template `geometric_distribution` models a random distribution . The distribution produces positive integers which are the number of bernoulli trials with probability p required to get one that fails.

For the geometric distribution, $p(i) = p(1-p)^i$.

Warning

This distribution has been updated to match the C++ standard. Its behavior has changed from the original boost::geometric_distribution. A backwards compatible wrapper is provided in namespace boost.

`geometric_distribution` public construct/copy/destruct

1.
```
explicit geometric_distribution(const RealType & p = 0.5);
```

   Contructs a new `geometric_distribution` with the paramter p.

   Requires: 0 < p < 1

2.
```
explicit geometric_distribution(const param_type & param);
```

   Constructs a new `geometric_distribution` from its parameters.

`geometric_distribution` public member functions

1.
```
RealType p() const;
```

   Returns: the distribution parameter p

2.
```
IntType min() const;
```

   Returns the smallest value that the distribution can produce.

3.
```
IntType max() const;
```

   Returns the largest value that the distribution can produce.

4.
```
param_type param() const;
```

   Returns the parameters of the distribution.

5.
```
void param(const param_type & param);
```

   Sets the parameters of the distribution.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6.
```
void reset();
```

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

7.
```
template<typename Engine> result_type operator()(Engine & eng) const;
```

Returns a random variate distributed according to the `geometric_distribution`.

8.
```
template<typename Engine> 
  result_type operator()(Engine & eng, const param_type & param) const;
```

Returns a random variate distributed according to the geometric distribution with parameters specified by param.

`geometric_distribution` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const geometric_distribution & gd);
```

Writes the distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const geometric_distribution & gd);
```

Reads the distribution from a `std::istream`.

3.
```
friend bool operator==(const geometric_distribution & lhs, 
                       const geometric_distribution & rhs);
```

Returns true if the two distributions will produce identical sequences of values given equal generators.

4.
```
friend bool operator!=(const geometric_distribution & lhs, 
                       const geometric_distribution & rhs);
```

Returns true if the two distributions may produce different sequences of values given equal generators.

Class param_type

boost::random::geometric_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/geometric_distribution.hpp>



class param_type {
public:
  // types
  typedef geometric_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 0.5);

  // public member functions
  RealType p() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
```
explicit param_type(RealType p = 0.5);
```

Constructs the parameters with p.

`param_type` public member functions

1.
```
RealType p() const;
```

Returns the p parameter of the distribution.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

Writes the parameters to a std::ostream.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const param_type & param);
```

Reads the parameters from a std::istream.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are equal.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are different.

Header <boost/random/independent_bits.hpp>

```
namespace boost {
  namespace random {
    template<typename Engine, std::size_t w, typename UIntType> 
      class independent_bits_engine;
  }
}
```

Class template independent_bits_engine

boost::random::independent_bits_engine

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/independent_bits.hpp>

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

  // construct/copy/destruct
  independent_bits_engine();
  explicit independent_bits_engine(result_type);
  template<typename SeedSeq> explicit independent_bits_engine(SeedSeq &);
  independent_bits_engine(const base_type &);
  template<typename It> independent_bits_engine(It &, It);

  // public static functions
  static result_type min();
  static result_type max();

  // public member functions
  void seed();
  void seed(result_type);
  template<typename SeedSeq> void seed(SeedSeq &);
  template<typename It> void seed(It &, It);
  result_type operator()();
  template<typename Iter> void generate(Iter, Iter);
  void discard(boost::uintmax_t);
  const base_type & base() const;

  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, 
               const independent_bits_engine &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
    operator>>(std::basic_istream< CharT, Traits > &, 
               const independent_bits_engine &);
  friend bool operator==(const independent_bits_engine &, 
                         const independent_bits_engine &);
  friend bool operator!=(const independent_bits_engine &, 
                         const independent_bits_engine &);

  // public data members
  static const bool has_fixed_range;
};
```

Description

An instantiation of class template `independent_bits_engine` model a pseudo-random number generator . It generates random numbers distributed between [0, 2^w) by combining one or more invocations of the base engine.

Requires: 0 < w <= std::numeric_limits<UIntType>::digits

`independent_bits_engine` public construct/copy/destruct

1.
```
independent_bits_engine();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Constructs an `independent_bits_engine` using the default constructor of the base generator.

2.
```
explicit independent_bits_engine(result_type seed);
```

Constructs an `independent_bits_engine`, using seed as the constructor argument for both base generators.

3.
```
template<typename SeedSeq> explicit independent_bits_engine(SeedSeq & seq);
```

Constructs an `independent_bits_engine`, using seq as the constructor argument for the base generator.

4.
```
independent_bits_engine(const base_type & base_arg);
```

Constructs an `independent_bits_engine` by copying `base`.

5.
```
template<typename It> independent_bits_engine(It & first, It last);
```

Contructs an `independent_bits_engine` with values from the range defined by the input iterators first and last. first will be modified to point to the element after the last one used.

Throws: `std::invalid_argument` if the input range is too small.

Exception Safety: Basic

`independent_bits_engine` public static functions

1.
```
static result_type min();
```

Returns the smallest value that the generator can produce.

2.
```
static result_type max();
```

Returns the largest value that the generator can produce.

`independent_bits_engine` public member functions

1.
```
void seed();
```

Seeds an `independent_bits_engine` using the default seed of the base generator.

2.
```
void seed(result_type seed);
```

Seeds an `independent_bits_engine`, using `seed` as the seed for the base generator.

3.
```
template<typename SeedSeq> void seed(SeedSeq & seq);
```

Seeds an `independent_bits_engine`, using `seq` to seed the base generator.

4.
```
template<typename It> void seed(It & first, It last);
```

Seeds an `independent_bits_engine` with values from the range defined by the input iterators first and last. first will be modified to point to the element after the last one used.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Throws: `std::invalid_argument` if the input range is too small.

Exception Safety: Basic

5.
   ```
   result_type operator()();
   ```

   Returns the next value of the generator.

6.
   ```
   template<typename Iter> void generate(Iter first, Iter last);
   ```

   Fills a range with random values

7.
   ```
   void discard(boost::uintmax_t z);
   ```

   Advances the state of the generator by `z`.

8.
   ```
   const base_type & base() const;
   ```

`independent_bits_engine` friend functions

1.
   ```
   template<typename CharT, typename Traits> 
     friend std::basic_ostream< CharT, Traits > & 
     operator<<(std::basic_ostream< CharT, Traits > & os, 
                const independent_bits_engine & r);
   ```

   Writes the textual representation if the generator to a `std::ostream`. The textual representation of the engine is the textual representation of the base engine.

2.
   ```
   template<typename CharT, typename Traits> 
     friend std::basic_istream< CharT, Traits > & 
     operator>>(std::basic_istream< CharT, Traits > & is, 
                const independent_bits_engine & r);
   ```

   Reads the state of an `independent_bits_engine` from a `std::istream`.

3.
   ```
   friend bool operator==(const independent_bits_engine & x, 
                          const independent_bits_engine & y);
   ```

   Returns: true iff the two `independent_bits_engines` will produce the same sequence of values.

4.
   ```
   friend bool operator!=(const independent_bits_engine & lhs, 
                          const independent_bits_engine & rhs);
   ```

   Returns: true iff the two `independent_bits_engines` will produce different sequences of values.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/random/inversive_congruential.hpp>

```
namespace boost {
  namespace random {
    template<typename IntType, IntType a, IntType b, IntType p> 
      class inversive_congruential_engine;
    typedef inversive_congruential_en↵
gine< uint32_t, 9102, 2147483647-36884165, 2147483647 > hellekalek1995;
  }
}
```

Class template inversive_congruential_engine

boost::random::inversive_congruential_engine

Synopsis

```
// In header: <boost/random/inversive_congruential.hpp>

template<typename IntType, IntType a, IntType b, IntType p> 
class inversive_congruential_engine {
public:
  // types
  typedef IntType result_type;

  // construct/copy/destruct
  inversive_congruential_engine();
  explicit inversive_congruential_engine(IntType);
  template<typename SeedSeq> explicit inversive_congruential_engine(SeedSeq &);
  template<typename It> inversive_congruential_engine(It &, It);

  // public static functions
  static result_type min();
  static result_type max();

  // public member functions
  void seed();
  void seed(IntType);
  template<typename SeedSeq> void seed(SeedSeq &);
  template<typename It> void seed(It &, It);
  IntType operator()();
  template<typename Iter> void generate(Iter, Iter);
  void discard(boost::uintmax_t);

  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, 
               const inversive_congruential_engine &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
    operator>>(std::basic_istream< CharT, Traits > &, 
               const inversive_congruential_engine &);
  friend bool operator==(const inversive_congruential_engine &, 
                         const inversive_congruential_engine &);
  friend bool operator!=(const inversive_congruential_engine &, 
                         const inversive_congruential_engine &);

  // public data members
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  static const bool has_fixed_range;
  static const result_type multiplier;
  static const result_type increment;
  static const result_type modulus;
  static const IntType default_seed;
};
```

Description

Instantiations of class template `inversive_congruential_engine` model a pseudo-random number generator . It uses the inversive congruential algorithm (ICG) described in

> "Inversive pseudorandom number generators: concepts, results and links", Peter Hellekalek, In: "Proceedings of the 1995 Winter Simulation Conference", C. Alexopoulos, K. Kang, W.R. Lilegdon, and D. Goldsman (editors), 1995, pp. 255-262. ftp://random.mat.sbg.ac.at/pub/data/wsc95.ps

The output sequence is defined by x(n+1) = (a*inv(x(n)) - b) (mod p), where x(0), a, b, and the prime number p are parameters of the generator. The expression inv(k) denotes the multiplicative inverse of k in the field of integer numbers modulo p, with inv(0) := 0.

The template parameter IntType shall denote a signed integral type large enough to hold p; a, b, and p are the parameters of the generators. The template parameter val is the validation value checked by validation.

> **Note**
>
> The implementation currently uses the Euclidian Algorithm to compute the multiplicative inverse. Therefore, the inversive generators are about 10-20 times slower than the others (see section"performance"). However, the paper talks of only 3x slowdown, so the Euclidian Algorithm is probably not optimal for calculating the multiplicative inverse.

`inversive_congruential_engine` public construct/copy/destruct

1.
   ```
   inversive_congruential_engine();
   ```

   Constructs an `inversive_congruential_engine`, seeding it with the default seed.

2.
   ```
   explicit inversive_congruential_engine(IntType x0);
   ```

   Constructs an `inversive_congruential_engine`, seeding it with `x0`.

3.
   ```
   template<typename SeedSeq>
     explicit inversive_congruential_engine(SeedSeq & seq);
   ```

   Constructs an `inversive_congruential_engine`, seeding it with values produced by a call to `seq.generate()`.

4.
   ```
   template<typename It> inversive_congruential_engine(It & first, It last);
   ```

   Constructs an `inversive_congruential_engine`, seeds it with values taken from the itrator range [first, last), and adjusts first to point to the element after the last one used. If there are not enough elements, throws `std::invalid_argument`.

   first and last must be input iterators.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`inversive_congruential_engine` public static functions

1.
```
static result_type min();
```

2.
```
static result_type max();
```

`inversive_congruential_engine` public member functions

1.
```
void seed();
```

   Calls seed(default_seed)

2.
```
void seed(IntType x0);
```

   If c mod m is zero and x0 mod m is zero, changes the current value of the generator to 1. Otherwise, changes it to x0 mod m. If c is zero, distinct seeds in the range [1,m) will leave the generator in distinct states. If c is not zero, the range is [0,m).

3.
```
template<typename SeedSeq> void seed(SeedSeq & seq);
```

   Seeds an `inversive_congruential_engine` using values from a SeedSeq.

4.
```
template<typename It> void seed(It & first, It last);
```

   seeds an `inversive_congruential_engine` with values taken from the itrator range [first, last) and adjusts `first` to point to the element after the last one used. If there are not enough elements, throws `std::invalid_argument`.

   `first` and `last` must be input iterators.

5.
```
IntType operator()();
```

   Returns the next output of the generator.

6.
```
template<typename Iter> void generate(Iter first, Iter last);
```

   Fills a range with random values

7.
```
void discard(boost::uintmax_t z);
```

   Advances the state of the generator by `z`.

`inversive_congruential_engine` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const inversive_congruential_engine & x);
```

   Writes the textual representation of the generator to a `std::ostream`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const inversive_congruential_engine & x);
```

Reads the textual representation of the generator from a `std::istream`.

3.
```
friend bool operator==(const inversive_congruential_engine & x, 
                       const inversive_congruential_engine & y);
```

Returns true if the two generators will produce identical sequences of outputs.

4.
```
friend bool operator!=(const inversive_congruential_engine & lhs, 
                       const inversive_congruential_engine & rhs);
```

Returns true if the two generators will produce different sequences of outputs.

Type definition hellekalek1995

hellekalek1995

Synopsis

```
// In header: <boost/random/inversive_congruential.hpp>


typedef inversive_congruential_en↵
gine< uint32_t, 9102, 2147483647-36884165, 2147483647 > hellekalek1995;
```

Description

The specialization hellekalek1995 was suggested in

> "Inversive pseudorandom number generators: concepts, results and links", Peter Hellekalek, In: "Proceedings of the 1995 Winter Simulation Conference", C. Alexopoulos, K. Kang, W.R. Lilegdon, and D. Goldsman (editors), 1995, pp. 255-262. ftp://random.mat.sbg.ac.at/pub/data/wsc95.ps

Header <boost/random/lagged_fibonacci.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType, int w, unsigned int p, unsigned int q> 
      class lagged_fibonacci_01_engine;
    template<typename UIntType, int w, unsigned int p, unsigned int q> 
      class lagged_fibonacci_engine;
    typedef lagged_fibonacci_01_engine< double, 48, 607, 273 > lagged_fibonacci607;
    typedef lagged_fibonacci_01_engine< double, 48, 1279, 418 > lagged_fibonacci1279;
    typedef lagged_fibonacci_01_engine< double, 48, 2281, 1252 > lagged_fibonacci2281;
    typedef lagged_fibonacci_01_engine< double, 48, 3217, 576 > lagged_fibonacci3217;
    typedef lagged_fibonacci_01_engine< double, 48, 4423, 2098 > lagged_fibonacci4423;
    typedef lagged_fibonacci_01_engine< double, 48, 9689, 5502 > lagged_fibonacci9689;
    typedef lagged_fibonacci_01_engine< double, 48, 19937, 9842 > lagged_fibonacci19937;
    typedef lagged_fibonacci_01_engine< double, 48, 23209, 13470 > lagged_fibonacci23209;
    typedef lagged_fibonacci_01_engine< double, 48, 44497, 21034 > lagged_fibonacci44497;
  }
}
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Class template lagged_fibonacci_01_engine

boost::random::lagged_fibonacci_01_engine

Synopsis

```
// In header: <boost/random/lagged_fibonacci.hpp>

template<typename RealType, int w, unsigned int p, unsigned int q>
class lagged_fibonacci_01_engine {
public:
  // types
  typedef RealType result_type;

  // construct/copy/destruct
  lagged_fibonacci_01_engine();
  explicit lagged_fibonacci_01_engine(uint32_t);
  template<typename SeedSeq> explicit lagged_fibonacci_01_engine(SeedSeq &);
  template<typename It> lagged_fibonacci_01_engine(It &, It);

  // public member functions
  void seed();
  void seed(boost::uint32_t);
  template<typename SeedSeq> void seed(SeedSeq &);
  template<typename It> void seed(It &, It);
  result_type operator()();
  template<typename Iter> void generate(Iter, Iter);
  void discard(boost::uintmax_t);

  // public static functions
  static result_type min();
  static result_type max();

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const lagged_fibonacci_01_engine &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
               const lagged_fibonacci_01_engine &);
  friend bool operator==(const lagged_fibonacci_01_engine &,
                         const lagged_fibonacci_01_engine &);
  friend bool operator!=(const lagged_fibonacci_01_engine &,
                         const lagged_fibonacci_01_engine &);

  // public data members
  static const bool has_fixed_range;
  static const int word_size;
  static const unsigned int long_lag;
  static const unsigned int short_lag;
  static const boost::uint32_t default_seed;
};
```

Description

Instantiations of class template `lagged_fibonacci_01` model a pseudo-random number generator . It uses a lagged Fibonacci algorithm with two lags `p` and `q`, evaluated in floating-point arithmetic: x(i) = x(i-p) + x(i-q) (mod 1) with p > q. See

> "Uniform random number generators for supercomputers", Richard Brent, Proc. of Fifth Australian Supercomputer Conference, Melbourne, Dec. 1992, pp. 704-706.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note

The quality of the generator crucially depends on the choice of the parameters. User code should employ one of the sensibly parameterized generators such as lagged_fibonacci607 instead.

The generator requires considerable amounts of memory for the storage of its state array. For example, lagged_fibonacci607 requires about 4856 bytes and lagged_fibonacci44497 requires about 350 KBytes.

`lagged_fibonacci_01_engine` public construct/copy/destruct

1.
```
lagged_fibonacci_01_engine();
```

   Constructs a `lagged_fibonacci_01` generator and calls `seed()`.

2.
```
explicit lagged_fibonacci_01_engine(uint32_t value);
```

   Constructs a `lagged_fibonacci_01` generator and calls `seed(value)`.

3.
```
template<typename SeedSeq> explicit lagged_fibonacci_01_engine(SeedSeq & seq);
```

   Constructs a `lagged_fibonacci_01` generator and calls `seed(gen)`.

4.
```
template<typename It> lagged_fibonacci_01_engine(It & first, It last);
```

`lagged_fibonacci_01_engine` public member functions

1.
```
void seed();
```

   Calls seed(default_seed).

2.
```
void seed(boost::uint32_t value);
```

   Constructs a `minstd_rand0` generator with the constructor parameter value and calls seed with it. Distinct seeds in the range [1, 2147483647) will produce generators with different states. Other seeds will be equivalent to some seed within this range. See `linear_congruential_engine` for details.

3.
```
template<typename SeedSeq> void seed(SeedSeq & seq);
```

   Seeds this `lagged_fibonacci_01_engine` using values produced by `seq.generate`.

4.
```
template<typename It> void seed(It & first, It last);
```

   Seeds this `lagged_fibonacci_01_engine` using values from the iterator range [first, last). If there are not enough elements in the range, throws `std::invalid_argument`.

5.
```
result_type operator()();
```

   Returns the next value of the generator.

6.
```
template<typename Iter> void generate(Iter first, Iter last);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Fills a range with random values

7.
```
void discard(boost::uintmax_t z);
```

Advances the state of the generator by `z`.

`lagged_fibonacci_01_engine` public static functions

1.
```
static result_type min();
```

Returns the smallest value that the generator can produce.

2.
```
static result_type max();
```

Returns the upper bound of the generators outputs.

`lagged_fibonacci_01_engine` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const lagged_fibonacci_01_engine & f);
```

Writes the textual representation of the generator to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const lagged_fibonacci_01_engine & f);
```

Reads the textual representation of the generator from a `std::istream`.

3.
```
friend bool operator==(const lagged_fibonacci_01_engine & x_, 
                       const lagged_fibonacci_01_engine & y_);
```

Returns true if the two generators will produce identical sequences of outputs.

4.
```
friend bool operator!=(const lagged_fibonacci_01_engine & lhs, 
                       const lagged_fibonacci_01_engine & rhs);
```

Returns true if the two generators will produce different sequences of outputs.

Class template lagged_fibonacci_engine

boost::random::lagged_fibonacci_engine

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/lagged_fibonacci.hpp>

template<typename UIntType, int w, unsigned int p, unsigned int q>
class lagged_fibonacci_engine {
public:
  // types
  typedef UIntType result_type;

  // construct/copy/destruct
  lagged_fibonacci_engine();
  explicit lagged_fibonacci_engine(UIntType);
  template<typename SeedSeq> explicit lagged_fibonacci_engine(SeedSeq &);
  template<typename It> lagged_fibonacci_engine(It &, It);

  // public static functions
  static result_type min();
  static result_type max();

  // public member functions
  void seed();
  void seed(UIntType);
  template<typename SeedSeq> void seed(SeedSeq &);
  template<typename It> void seed(It &, It);
  result_type operator()();
  template<typename Iter> void generate(Iter, Iter);
  void discard(boost::uintmax_t);

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const lagged_fibonacci_engine &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
               const lagged_fibonacci_engine &);
  friend bool operator==(const lagged_fibonacci_engine &,
                         const lagged_fibonacci_engine &);
  friend bool operator!=(const lagged_fibonacci_engine &,
                         const lagged_fibonacci_engine &);

  // public data members
  static const bool has_fixed_range;
  static const int word_size;
  static const unsigned int long_lag;
  static const unsigned int short_lag;
  static const UIntType default_seed;
};
```

Description

Instantiations of class template lagged_fibonacci_engine model a pseudo-random number generator . It uses a lagged Fibonacci algorithm with two lags p and q: x(i) = x(i-p) + x(i-q) (mod 2^w) with p > q.

`lagged_fibonacci_engine` public construct/copy/destruct

1.
```
lagged_fibonacci_engine();
```

   Creates a new `lagged_fibonacci_engine` and calls `seed()`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
explicit lagged_fibonacci_engine(UIntType value);
```

   Creates a new `lagged_fibonacci_engine` and calls `seed(value)`.

3.
```
template<typename SeedSeq> explicit lagged_fibonacci_engine(SeedSeq & seq);
```

   Creates a new `lagged_fibonacci_engine` and calls `seed(seq)`.

4.
```
template<typename It> lagged_fibonacci_engine(It & first, It last);
```

   Creates a new `lagged_fibonacci_engine` and calls `seed(first, last)`.

`lagged_fibonacci_engine` public static functions

1.
```
static result_type min();
```

   Returns the smallest value that the generator can produce.

2.
```
static result_type max();
```

   Returns the largest value that the generator can produce.

`lagged_fibonacci_engine` public member functions

1.
```
void seed();
```

   Calls `seed(default_seed)`.

2.
```
void seed(UIntType value);
```

   Sets the state of the generator to values produced by a `minstd_rand0` generator.

3.
```
template<typename SeedSeq> void seed(SeedSeq & seq);
```

   Sets the state of the generator using values produced by seq.

4.
```
template<typename It> void seed(It & first, It last);
```

   Sets the state of the generator to values from the iterator range [first, last). If there are not enough elements in the range [first, last) throws `std::invalid_argument`.

5.
```
result_type operator()();
```

   Returns the next value of the generator.

6.
```
template<typename Iter> void generate(Iter first, Iter last);
```

   Fills a range with random values

7.
```
void discard(boost::uintmax_t z);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Advances the state of the generator by `z`.

`lagged_fibonacci_engine` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const lagged_fibonacci_engine & f);
```

   Writes the textual representation of the generator to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const lagged_fibonacci_engine & f);
```

   Reads the textual representation of the generator from a `std::istream`.

3.
```
friend bool operator==(const lagged_fibonacci_engine & x_,
                       const lagged_fibonacci_engine & y_);
```

   Returns true if the two generators will produce identical sequences of outputs.

4.
```
friend bool operator!=(const lagged_fibonacci_engine & lhs,
                       const lagged_fibonacci_engine & rhs);
```

   Returns true if the two generators will produce different sequences of outputs.

Type definition lagged_fibonacci607

lagged_fibonacci607

Synopsis

```
// In header: <boost/random/lagged_fibonacci.hpp>


typedef lagged_fibonacci_01_engine< double, 48, 607, 273 > lagged_fibonacci607;
```

Description

The specializations lagged_fibonacci607 ... lagged_fibonacci44497 use well tested lags.

See

> "On the Periods of Generalized Fibonacci Recurrences", Richard P. Brent Computer Sciences Laboratory Australian National University, December 1992

The lags used here can be found in

> "Uniform random number generators for supercomputers", Richard Brent, Proc. of Fifth Australian Supercomputer Conference, Melbourne, Dec. 1992, pp. 704-706.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Type definition lagged_fibonacci1279

lagged_fibonacci1279

Synopsis

```
// In header: <boost/random/lagged_fibonacci.hpp>


typedef lagged_fibonacci_01_engine< double, 48, 1279, 418 > lagged_fibonacci1279;
```

Description

The specializations lagged_fibonacci607 ... lagged_fibonacci44497 use well tested lags.

See

> "On the Periods of Generalized Fibonacci Recurrences", Richard P. Brent Computer Sciences Laboratory Australian National University, December 1992

The lags used here can be found in

> "Uniform random number generators for supercomputers", Richard Brent, Proc. of Fifth Australian Supercomputer Conference, Melbourne, Dec. 1992, pp. 704-706.

Type definition lagged_fibonacci2281

lagged_fibonacci2281

Synopsis

```
// In header: <boost/random/lagged_fibonacci.hpp>


typedef lagged_fibonacci_01_engine< double, 48, 2281, 1252 > lagged_fibonacci2281;
```

Description

The specializations lagged_fibonacci607 ... lagged_fibonacci44497 use well tested lags.

See

> "On the Periods of Generalized Fibonacci Recurrences", Richard P. Brent Computer Sciences Laboratory Australian National University, December 1992

The lags used here can be found in

> "Uniform random number generators for supercomputers", Richard Brent, Proc. of Fifth Australian Supercomputer Conference, Melbourne, Dec. 1992, pp. 704-706.

Type definition lagged_fibonacci3217

lagged_fibonacci3217

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/lagged_fibonacci.hpp>


typedef lagged_fibonacci_01_engine< double, 48, 3217, 576 > lagged_fibonacci3217;
```

Description

The specializations lagged_fibonacci607 ... lagged_fibonacci44497 use well tested lags.

See

> "On the Periods of Generalized Fibonacci Recurrences", Richard P. Brent Computer Sciences Laboratory Australian National University, December 1992

The lags used here can be found in

> "Uniform random number generators for supercomputers", Richard Brent, Proc. of Fifth Australian Supercomputer Conference, Melbourne, Dec. 1992, pp. 704-706.

Type definition lagged_fibonacci4423

lagged_fibonacci4423

Synopsis

```
// In header: <boost/random/lagged_fibonacci.hpp>


typedef lagged_fibonacci_01_engine< double, 48, 4423, 2098 > lagged_fibonacci4423;
```

Description

The specializations lagged_fibonacci607 ... lagged_fibonacci44497 use well tested lags.

See

> "On the Periods of Generalized Fibonacci Recurrences", Richard P. Brent Computer Sciences Laboratory Australian National University, December 1992

The lags used here can be found in

> "Uniform random number generators for supercomputers", Richard Brent, Proc. of Fifth Australian Supercomputer Conference, Melbourne, Dec. 1992, pp. 704-706.

Type definition lagged_fibonacci9689

lagged_fibonacci9689

Synopsis

```
// In header: <boost/random/lagged_fibonacci.hpp>


typedef lagged_fibonacci_01_engine< double, 48, 9689, 5502 > lagged_fibonacci9689;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

The specializations lagged_fibonacci607 ... lagged_fibonacci44497 use well tested lags.

See

> "On the Periods of Generalized Fibonacci Recurrences", Richard P. Brent Computer Sciences Laboratory Australian National University, December 1992

The lags used here can be found in

> "Uniform random number generators for supercomputers", Richard Brent, Proc. of Fifth Australian Supercomputer Conference, Melbourne, Dec. 1992, pp. 704-706.

Type definition lagged_fibonacci19937

lagged_fibonacci19937

Synopsis

```
// In header: <boost/random/lagged_fibonacci.hpp>


typedef lagged_fibonacci_01_engine< double, 48, 19937, 9842 > lagged_fibonacci19937;
```

Description

The specializations lagged_fibonacci607 ... lagged_fibonacci44497 use well tested lags.

See

> "On the Periods of Generalized Fibonacci Recurrences", Richard P. Brent Computer Sciences Laboratory Australian National University, December 1992

The lags used here can be found in

> "Uniform random number generators for supercomputers", Richard Brent, Proc. of Fifth Australian Supercomputer Conference, Melbourne, Dec. 1992, pp. 704-706.

Type definition lagged_fibonacci23209

lagged_fibonacci23209

Synopsis

```
// In header: <boost/random/lagged_fibonacci.hpp>


typedef lagged_fibonacci_01_engine< double, 48, 23209, 13470 > lagged_fibonacci23209;
```

Description

The specializations lagged_fibonacci607 ... lagged_fibonacci44497 use well tested lags.

See

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

> "On the Periods of Generalized Fibonacci Recurrences", Richard P. Brent Computer Sciences Laboratory Australian National University, December 1992

The lags used here can be found in

> "Uniform random number generators for supercomputers", Richard Brent, Proc. of Fifth Australian Supercomputer Conference, Melbourne, Dec. 1992, pp. 704-706.

Type definition lagged_fibonacci44497

lagged_fibonacci44497

Synopsis

```
// In header: <boost/random/lagged_fibonacci.hpp>


typedef lagged_fibonacci_01_engine< double, 48, 44497, 21034 > lagged_fibonacci44497;
```

Description

The specializations lagged_fibonacci607 ... lagged_fibonacci44497 use well tested lags.

See

> "On the Periods of Generalized Fibonacci Recurrences", Richard P. Brent Computer Sciences Laboratory Australian National University, December 1992

The lags used here can be found in

> "Uniform random number generators for supercomputers", Richard Brent, Proc. of Fifth Australian Supercomputer Conference, Melbourne, Dec. 1992, pp. 704-706.

Header <boost/random/laplace_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class laplace_distribution;
  }
}
```

Class template laplace_distribution

boost::random::laplace_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/laplace_distribution.hpp>

template<typename RealType = double>
class laplace_distribution {
public:
  // types
  typedef RealType result_type;
  typedef RealType input_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef laplace_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 0.0, RealType = 1.0);

    // public member functions
    RealType mean() const;
    RealType beta() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit laplace_distribution(RealType = 0.0, RealType = 1.0);
  explicit laplace_distribution(const param_type &);

  // public member functions
  template<typename URNG> RealType operator()(URNG &) const;
  template<typename URNG>
    RealType operator()(URNG &, const param_type &) const;
  RealType mean() const;
  RealType beta() const;
  RealType min() const;
  RealType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const laplace_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
                   const laplace_distribution &);
  friend bool operator==(const laplace_distribution &,
                         const laplace_distribution &);
  friend bool operator!=(const laplace_distribution &,
                         const laplace_distribution &);
};
```

Description

The laplace distribution is a real-valued distribution with two parameters, mean and beta.

It has $p(x) = \frac{e^{-\frac{|x-\mu|}{\beta}}}{2\beta}$.

`laplace_distribution` public construct/copy/destruct

1.
   ```
   explicit laplace_distribution(RealType mean = 0.0, RealType beta = 1.0);
   ```

   Constructs an `laplace_distribution` from its "mean" and "beta" parameters.

2.
   ```
   explicit laplace_distribution(const param_type & param);
   ```

   Constructs an `laplace_distribution` from its parameters.

`laplace_distribution` public member functions

1.
   ```
   template<typename URNG> RealType operator()(URNG & urng) const;
   ```

   Returns a random variate distributed according to the laplace distribution.

2.
   ```
   template<typename URNG> 
     RealType operator()(URNG & urng, const param_type & param) const;
   ```

   Returns a random variate distributed accordint to the laplace distribution with parameters specified by `param`.

3.
   ```
   RealType mean() const;
   ```

   Returns the "mean" parameter of the distribution.

4.
   ```
   RealType beta() const;
   ```

   Returns the "beta" parameter of the distribution.

5.
   ```
   RealType min() const;
   ```

   Returns the smallest value that the distribution can produce.

6.
   ```
   RealType max() const;
   ```

   Returns the largest value that the distribution can produce.

7.
   ```
   param_type param() const;
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns the parameters of the distribution.

8.
```
void param(const param_type & param);
```

Sets the parameters of the distribution.

9.
```
void reset();
```

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

`laplace_distribution` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const laplace_distribution & wd);
```

Writes an `laplace_distribution` to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const laplace_distribution & wd);
```

Reads an `laplace_distribution` from a `std::istream`.

3.
```
friend bool operator==(const laplace_distribution & lhs, 
                       const laplace_distribution & rhs);
```

Returns true if the two instances of `laplace_distribution` will return identical sequences of values given equal generators.

4.
```
friend bool operator!=(const laplace_distribution & lhs, 
                       const laplace_distribution & rhs);
```

Returns true if the two instances of `laplace_distribution` will return different sequences of values given equal generators.

Class param_type

boost::random::laplace_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/laplace_distribution.hpp>



class param_type {
public:
  // types
  typedef laplace_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 0.0, RealType = 1.0);

  // public member functions
  RealType mean() const;
  RealType beta() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
```
explicit param_type(RealType mean = 0.0, RealType beta = 1.0);
```

   Constructs a `param_type` from the "mean" and "beta" parameters of the distribution.

`param_type` public member functions

1.
```
RealType mean() const;
```

   Returns the "mean" parameter of the distribtuion.

2.
```
RealType beta() const;
```

   Returns the "beta" parameter of the distribution.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

   Writes a `param_type` to a `std::ostream`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const param_type & param);
```

Reads a `param_type` from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are the same.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are the different.

Header <boost/random/linear_congruential.hpp>

```
namespace boost {
  namespace random {
    template<typename IntType, IntType a, IntType c, IntType m>
      class linear_congruential_engine;
    class rand48;
    typedef linear_congruential_engine< uint32_t, 16807, 0, 2147483647 > minstd_rand0;
    typedef linear_congruential_engine< uint32_t, 48271, 0, 2147483647 > minstd_rand;
  }
}
```

Class template linear_congruential_engine

boost::random::linear_congruential_engine

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/linear_congruential.hpp>

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

  // construct/copy/destruct
  linear_congruential_engine();
  explicit linear_congruential_engine(IntType);
  template<typename SeedSeq> explicit linear_congruential_engine(SeedSeq &);
  template<typename It> linear_congruential_engine(It &, It);

  // public member functions
  void seed();
  void seed(IntType);
  template<typename SeedSeq> void seed(SeedSeq &);
  template<typename It> void seed(It &, It);
  IntType operator()();
  template<typename Iter> void generate(Iter, Iter);
  void discard(boost::uintmax_t);

  // public static functions
  static result_type min();
  static result_type max();

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const linear_congruential_engine &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
               linear_congruential_engine &);

  // public data members
  static const bool has_fixed_range;
  static const IntType multiplier;
  static const IntType increment;
  static const IntType modulus;
  static const IntType default_seed;
};
```

Description

Instantiations of class template linear_congruential_engine model a pseudo-random number generator . Linear congruential pseudo-random number generators are described in:

> "Mathematical methods in large-scale computing units", D. H. Lehmer, Proc. 2nd Symposium on Large-Scale Digital Calculating Machines, Harvard University Press, 1951, pp. 141-146

Let x(n) denote the sequence of numbers returned by some pseudo-random number generator. Then for the linear congruential generator, x(n+1) := (a * x(n) + c) mod m. Parameters for the generator are x(0), a, c, m. The template parameter IntType shall denote an integral type. It must be large enough to hold values a, c, and m. The template parameters a and c must be smaller than m.

Note: The quality of the generator crucially depends on the choice of the parameters. User code should use one of the sensibly parameterized generators such as minstd_rand instead.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`linear_congruential_engine` public construct/copy/destruct

1.
```
linear_congruential_engine();
```

   Constructs a `linear_congruential_engine`, using the default seed

2.
```
explicit linear_congruential_engine(IntType x0);
```

   Constructs a `linear_congruential_engine`, seeding it with `x0`.

3.
```
template<typename SeedSeq> explicit linear_congruential_engine(SeedSeq & seq);
```

   Constructs a `linear_congruential_engine`, seeding it with values produced by a call to `seq.generate()`.

4.
```
template<typename It> linear_congruential_engine(It & first, It last);
```

   Constructs a `linear_congruential_engine` and seeds it with values taken from the itrator range [first, last) and adjusts first to point to the element after the last one used. If there are not enough elements, throws `std::invalid_argument`.

   first and last must be input iterators.

`linear_congruential_engine` public member functions

1.
```
void seed();
```

   Calls seed(default_seed)

2.
```
void seed(IntType x0);
```

   If c mod m is zero and x0 mod m is zero, changes the current value of the generator to 1. Otherwise, changes it to x0 mod m. If c is zero, distinct seeds in the range [1,m) will leave the generator in distinct states. If c is not zero, the range is [0,m).

3.
```
template<typename SeedSeq> void seed(SeedSeq & seq);
```

   Seeds a `linear_congruential_engine` using values from a SeedSeq.

4.
```
template<typename It> void seed(It & first, It last);
```

   seeds a `linear_congruential_engine` with values taken from the itrator range [first, last) and adjusts `first` to point to the element after the last one used. If there are not enough elements, throws `std::invalid_argument`.

   `first` and `last` must be input iterators.

5.
```
IntType operator()();
```

   Returns the next value of the `linear_congruential_engine`.

6.
```
template<typename Iter> void generate(Iter first, Iter last);
```

   Fills a range with random values

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

7.
```
void discard(boost::uintmax_t z);
```

Advances the state of the generator by `z`.

`linear_congruential_engine` public static functions

1.
```
static result_type min();
```

Returns the smallest value that the `linear_congruential_engine` can produce.

2.
```
static result_type max();
```

Returns the largest value that the `linear_congruential_engine` can produce.

`linear_congruential_engine` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const linear_congruential_engine & lcg);
```

Writes a `linear_congruential_engine` to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             linear_congruential_engine & lcg);
```

Reads a `linear_congruential_engine` from a `std::istream`.

Class rand48

boost::random::rand48

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/linear_congruential.hpp>


class rand48 {
public:
  // types
  typedef boost::uint32_t result_type;

  // construct/copy/destruct
  rand48();
  explicit rand48(result_type);
  template<typename SeedSeq> explicit rand48(SeedSeq &);
  template<typename It> rand48(It &, It);

  // public static functions
  static uint32_t min();
  static uint32_t max();

  // public member functions
  void seed();
  void seed(result_type);
  template<typename It> void seed(It &, It);
  template<typename SeedSeq> void seed(SeedSeq &);
  uint32_t operator()();
  void discard(boost::uintmax_t);
  template<typename Iter> void generate(Iter, Iter);

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const rand48 &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, rand48 &);
  friend bool operator==(const rand48 &, const rand48 &);
  friend bool operator!=(const rand48 &, const rand48 &);

  // public data members
  static const bool has_fixed_range;
};
```

Description

Class `rand48` models a pseudo-random number generator . It uses the linear congruential algorithm with the parameters a = 0x5DEECE66D, c = 0xB, m = 2**48. It delivers identical results to the `lrand48()` function available on some systems (assuming lcong48 has not been called).

It is only available on systems where `uint64_t` is provided as an integral type, so that for example static in-class constants and/or enum definitions with large `uint64_t` numbers work.

`rand48` public construct/copy/destruct

1.
```
rand48();
```

   Seeds the generator with the default seed.

2.
```
explicit rand48(result_type x0);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Constructs a `rand48` generator with x(0) := (x0 << 16) | 0x330e.

3.
```
template<typename SeedSeq> explicit rand48(SeedSeq & seq);
```

Seeds the generator with values produced by `seq.generate()`.

4.
```
template<typename It> rand48(It & first, It last);
```

Seeds the generator using values from an iterator range, and updates first to point one past the last value consumed.

`rand48` public static functions

1.
```
static uint32_t min();
```

Returns the smallest value that the generator can produce

2.
```
static uint32_t max();
```

Returns the largest value that the generator can produce

`rand48` public member functions

1.
```
void seed();
```

Seeds the generator with the default seed.

2.
```
void seed(result_type x0);
```

Changes the current value x(n) of the generator to (x0 << 16) | 0x330e.

3.
```
template<typename It> void seed(It & first, It last);
```

Seeds the generator using values from an iterator range, and updates first to point one past the last value consumed.

4.
```
template<typename SeedSeq> void seed(SeedSeq & seq);
```

Seeds the generator with values produced by `seq.generate()`.

5.
```
uint32_t operator()();
```

Returns the next value of the generator.

6.
```
void discard(boost::uintmax_t z);
```

Advances the state of the generator by `z`.

7.
```
template<typename Iter> void generate(Iter first, Iter last);
```

Fills a range with random values

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`rand48` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, const rand48 & r);
```

   Writes a `rand48` to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, rand48 & r);
```

   Reads a `rand48` from a `std::istream`.

3.
```
friend bool operator==(const rand48 & x, const rand48 & y);
```

   Returns true if the two generators will produce identical sequences of values.

4.
```
friend bool operator!=(const rand48 & x, const rand48 & y);
```

   Returns true if the two generators will produce different sequences of values.

Type definition minstd_rand0

minstd_rand0

Synopsis

```
// In header: <boost/random/linear_congruential.hpp>


typedef linear_congruential_engine< uint32_t, 16807, 0, 2147483647 > minstd_rand0;
```

Description

The specialization minstd_rand0 was originally suggested in

> A pseudo-random number generator for the System/360, P.A. Lewis, A.S. Goodman, J.M. Miller, IBM Systems Journal, Vol. 8, No. 2, 1969, pp. 136-146

It is examined more closely together with minstd_rand in

> "Random Number Generators: Good ones are hard to find", Stephen K. Park and Keith W. Miller, Communications of the ACM, Vol. 31, No. 10, October 1988, pp. 1192-1201

Type definition minstd_rand

minstd_rand

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/linear_congruential.hpp>


typedef linear_congruential_engine< uint32_t, 48271, 0, 2147483647 > minstd_rand;
```

Description

The specialization minstd_rand was suggested in

> "Random Number Generators: Good ones are hard to find", Stephen K. Park and Keith W. Miller, Communications of the ACM, Vol. 31, No. 10, October 1988, pp. 1192-1201

Header <boost/random/linear_feedback_shift.hpp>

```
namespace boost {
  namespace random {
    template<typename UIntType, int w, int k, int q, int s> 
      class linear_feedback_shift_engine;
  }
}
```

Class template linear_feedback_shift_engine

boost::random::linear_feedback_shift_engine

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/linear_feedback_shift.hpp>

template<typename UIntType, int w, int k, int q, int s>
class linear_feedback_shift_engine {
public:
  // types
  typedef UIntType result_type;

  // construct/copy/destruct
  linear_feedback_shift_engine();
  explicit linear_feedback_shift_engine(UIntType);
  template<typename SeedSeq> explicit linear_feedback_shift_engine(SeedSeq &);
  template<typename It> linear_feedback_shift_engine(It &, It);

  // public static functions
  static result_type min();
  static result_type max();

  // public member functions
  void seed();
  void seed(UIntType);
  template<typename SeedSeq> void seed(SeedSeq &);
  template<typename It> void seed(It &, It);
  result_type operator()();
  template<typename Iter> void generate(Iter, Iter);
  void discard(boost::uintmax_t);

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const linear_feedback_shift_engine &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
               const linear_feedback_shift_engine &);
  friend bool operator==(const linear_feedback_shift_engine &,
                         const linear_feedback_shift_engine &);
  friend bool operator!=(const linear_feedback_shift_engine &,
                         const linear_feedback_shift_engine &);

  // public data members
  static const bool has_fixed_range;
  static const int word_size;
  static const int exponent1;
  static const int exponent2;
  static const int step_size;
  static const UIntType default_seed;
};
```

Description

Instatiations of `linear_feedback_shift` model a pseudo-random number generator . It was originally proposed in

> "Random numbers generated by linear recurrence modulo two.", Tausworthe, R. C.(1965), Mathematics of Computation 19, 201-209.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`linear_feedback_shift_engine` public construct/copy/destruct

1.
   ```
   linear_feedback_shift_engine();
   ```

   Constructs a `linear_feedback_shift_engine`, using the default seed.

2.
   ```
   explicit linear_feedback_shift_engine(UIntType s0);
   ```

   Constructs a `linear_feedback_shift_engine`, seeding it with s0.

3.
   ```
   template<typename SeedSeq> 
     explicit linear_feedback_shift_engine(SeedSeq & seq);
   ```

   Constructs a `linear_feedback_shift_engine`, seeding it with seq.

4.
   ```
   template<typename It> linear_feedback_shift_engine(It & first, It last);
   ```

   Constructs a `linear_feedback_shift_engine`, seeding it with values from the range [first, last).

`linear_feedback_shift_engine` public static functions

1.
   ```
   static result_type min();
   ```

   Returns the smallest value that the generator can produce.

2.
   ```
   static result_type max();
   ```

   Returns the largest value that the generator can produce.

`linear_feedback_shift_engine` public member functions

1.
   ```
   void seed();
   ```

   Seeds a `linear_feedback_shift_engine` with the default seed.

2.
   ```
   void seed(UIntType s0);
   ```

   Seeds a `linear_feedback_shift_engine` with `s0`.

3.
   ```
   template<typename SeedSeq> void seed(SeedSeq & seq);
   ```

   Seeds a `linear_feedback_shift_engine` with values produced by `seq.generate()`.

4.
   ```
   template<typename It> void seed(It & first, It last);
   ```

   Seeds a `linear_feedback_shift_engine` with values from the range [first, last).

5.
   ```
   result_type operator()();
   ```

   Returns the next value of the generator.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6.

```
template<typename Iter> void generate(Iter first, Iter last);
```

Fills a range with random values

7.

```
void discard(boost::uintmax_t z);
```

Advances the state of the generator by `z`.

`linear_feedback_shift_engine` friend functions

1.

```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const linear_feedback_shift_engine & x);
```

Writes the textual representation of the generator to a `std::ostream`.

2.

```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const linear_feedback_shift_engine & x);
```

Reads the textual representation of the generator from a `std::istream`.

3.

```
friend bool operator==(const linear_feedback_shift_engine & x, 
                       const linear_feedback_shift_engine & y);
```

Returns true if the two generators will produce identical sequences of outputs.

4.

```
friend bool operator!=(const linear_feedback_shift_engine & lhs, 
                       const linear_feedback_shift_engine & rhs);
```

Returns true if the two generators will produce different sequences of outputs.

Header <boost/random/lognormal_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class lognormal_distribution;
  }
}
```

Class template lognormal_distribution

boost::random::lognormal_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/lognormal_distribution.hpp>

template<typename RealType = double>
class lognormal_distribution {
public:
  // types
  typedef normal_distribution< RealType >::input_type input_type;
  typedef RealType                                    result_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef lognormal_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 0.0, RealType = 1.0);

    // public member functions
    RealType m() const;
    RealType s() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit lognormal_distribution(RealType = 0.0, RealType = 1.0);
  explicit lognormal_distribution(const param_type &);

  // public member functions
  RealType m() const;
  RealType s() const;
  RealType min() const;
  RealType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();
  template<typename Engine> result_type operator()(Engine &);
  template<typename Engine>
    result_type operator()(Engine &, const param_type &);

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const lognormal_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
                const lognormal_distribution &);
  friend bool operator==(const lognormal_distribution &,
                         const lognormal_distribution &);
  friend bool operator!=(const lognormal_distribution &,
                         const lognormal_distribution &);
};
```

Description

Instantiations of class template lognormal_distribution model a random distribution . Such a distribution produces random numbers with $p(x) = \frac{1}{x s \sqrt{2\pi}} e^{\frac{-(\log(x)-m)^2}{2s^2}}$ for x > 0.

> **Warning**
>
> This distribution has been updated to match the C++ standard. Its behavior has changed from the original boost::lognormal_distribution. A backwards compatible version is provided in namespace boost.

`lognormal_distribution` public construct/copy/destruct

1.
   ```
   explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);
   ```

   Constructs a `lognormal_distribution`. `m` and `s` are the parameters of the distribution.

2.
   ```
   explicit lognormal_distribution(const param_type & param);
   ```

   Constructs a `lognormal_distribution` from its parameters.

`lognormal_distribution` public member functions

1.
   ```
   RealType m() const;
   ```

   Returns the m parameter of the distribution.

2.
   ```
   RealType s() const;
   ```

   Returns the s parameter of the distribution.

3.
   ```
   RealType min() const;
   ```

   Returns the smallest value that the distribution can produce.

4.
   ```
   RealType max() const;
   ```

   Returns the largest value that the distribution can produce.

5.
   ```
   param_type param() const;
   ```

   Returns the parameters of the distribution.

6.
   ```
   void param(const param_type & param);
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Sets the parameters of the distribution.

7.
```
void reset();
```

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

8.
```
template<typename Engine> result_type operator()(Engine & eng);
```

Returns a random variate distributed according to the lognormal distribution.

9.
```
template<typename Engine>
  result_type operator()(Engine & eng, const param_type & param);
```

Returns a random variate distributed according to the lognormal distribution with parameters specified by param.

`lognormal_distribution` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const lognormal_distribution & ld);
```

Writes the distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const lognormal_distribution & ld);
```

Reads the distribution from a `std::istream`.

3.
```
friend bool operator==(const lognormal_distribution & lhs,
                       const lognormal_distribution & rhs);
```

Returns true if the two distributions will produce identical sequences of values given equal generators.

4.
```
friend bool operator!=(const lognormal_distribution & lhs,
                       const lognormal_distribution & rhs);
```

Returns true if the two distributions may produce different sequences of values given equal generators.

Class param_type

boost::random::lognormal_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/lognormal_distribution.hpp>



class param_type {
public:
  // types
  typedef lognormal_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 0.0, RealType = 1.0);

  // public member functions
  RealType m() const;
  RealType s() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
```
explicit param_type(RealType m = 0.0, RealType s = 1.0);
```

Constructs the parameters of a `lognormal_distribution`.

`param_type` public member functions

1.
```
RealType m() const;
```

Returns the "m" parameter of the distribution.

2.
```
RealType s() const;
```

Returns the "s" parameter of the distribution.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

Writes the parameters to a std::ostream.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const param_type & param);
```

   Reads the parameters from a std::istream.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

   Returns true if the two sets of parameters are equal.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

   Returns true if the two sets of parameters are different.

Header <boost/random/mersenne_twister.hpp>

```
namespace boost {
  namespace random {
    template<typename UIntType, std::size_t w, std::size_t n, std::size_t m, 
             std::size_t r, UIntType a, std::size_t u, UIntType d, 
             std::size_t s, UIntType b, std::size_t t, UIntType c, 
             std::size_t l, UIntType f> 
      class mersenne_twister_engine;
    typedef mersenne_twister_en↵
gine< uint32_t, 32, 351, 175, 19, 0xccab8ee7, 11, 0xffffffff, 7, 0x31b6ab00, 15, 0xffe50000, 17, 1812433253 > mt11213b;
    typedef mersenne_twister_en↵
gine< uint32_t, 32, 624, 397, 31, 0x9908b0df, 11, 0xffffffff, 7, 0x9d2c5680, 15, 0xefc60000, 18, 1812433253 > mt19937;
    typedef mersenne_twister_en↵
gine< uint64_t, 64, 312, 156, 31, 0xb5026f5aa96619e9ull, 29, 0x5555555555555555ull, 17, 0x71d67fffeda60000ull, 37, 0xfff7eee000000000ull, 43, 6364136223846793005ull > mt19937_64;
  }
}
```

Class template mersenne_twister_engine

boost::random::mersenne_twister_engine

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/mersenne_twister.hpp>

template<typename UIntType, std::size_t w, std::size_t n, std::size_t m, 
         std::size_t r, UIntType a, std::size_t u, UIntType d, std::size_t s, 
         UIntType b, std::size_t t, UIntType c, std::size_t l, UIntType f> 
class mersenne_twister_engine {
public:
  // types
  typedef UIntType result_type;

  // construct/copy/destruct
  mersenne_twister_engine();
  explicit mersenne_twister_engine(UIntType);
  template<typename It> mersenne_twister_engine(It &, It);
  template<typename SeedSeq> explicit mersenne_twister_engine(SeedSeq &);

  // public member functions
  void seed();
  void seed(UIntType);
  template<typename SeeqSeq> void seed(SeeqSeq &);
  template<typename It> void seed(It &, It);
  result_type operator()();
  template<typename Iter> void generate(Iter, Iter);
  void discard(boost::uintmax_t);

  // public static functions
  static result_type min();
  static result_type max();

  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, 
               const mersenne_twister_engine &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
    operator>>(std::basic_istream< CharT, Traits > &, 
               mersenne_twister_engine &);
  friend bool operator==(const mersenne_twister_engine &, 
                         const mersenne_twister_engine &);
  friend bool operator!=(const mersenne_twister_engine &, 
                         const mersenne_twister_engine &);

  // public data members
  static const std::size_t word_size;
  static const std::size_t state_size;
  static const std::size_t shift_size;
  static const std::size_t mask_bits;
  static const UIntType xor_mask;
  static const std::size_t tempering_u;
  static const UIntType tempering_d;
  static const std::size_t tempering_s;
  static const UIntType tempering_b;
  static const std::size_t tempering_t;
  static const UIntType tempering_c;
  static const std::size_t tempering_l;
  static const UIntType initialization_multiplier;
  static const UIntType default_seed;
  static const UIntType parameter_a;
  static const std::size_t output_u;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  static const std::size_t output_s;
  static const UIntType output_b;
  static const std::size_t output_t;
  static const UIntType output_c;
  static const std::size_t output_l;
  static const bool has_fixed_range;
};
```

Description

Instantiations of class template mersenne_twister_engine model a pseudo-random number generator . It uses the algorithm described in

> "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator", Makoto Matsumoto and Takuji Nishimura, ACM Transactions on Modeling and Computer Simulation: Special Issue on Uniform Random Number Generation, Vol. 8, No. 1, January 1998, pp. 3-30.

> **Note**
>
> The boost variant has been implemented from scratch and does not derive from or use mt19937.c provided on the above WWW site. However, it was verified that both produce identical output.

The seeding from an integer was changed in April 2005 to address a weakness.

The quality of the generator crucially depends on the choice of the parameters. User code should employ one of the sensibly parameterized generators such as mt19937 instead.

The generator requires considerable amounts of memory for the storage of its state array. For example, mt11213b requires about 1408 bytes and mt19937 requires about 2496 bytes.

`mersenne_twister_engine` public construct/copy/destruct

1.
   ```
   mersenne_twister_engine();
   ```
   Constructs a `mersenne_twister_engine` and calls `seed()`.

2.
   ```
   explicit mersenne_twister_engine(UIntType value);
   ```
   Constructs a `mersenne_twister_engine` and calls `seed(value)`.

3.
   ```
   template<typename It> mersenne_twister_engine(It & first, It last);
   ```

4.
   ```
   template<typename SeedSeq> explicit mersenne_twister_engine(SeedSeq & seq);
   ```
   Constructs a `mersenne_twister_engine` and calls `seed(gen)`.

   > **Note**
   >
   > The copy constructor will always be preferred over the templated constructor.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`mersenne_twister_engine` public member functions

1.
```
void seed();
```

   Calls `seed(default_seed)`.

2.
```
void seed(UIntType value);
```

   Sets the state x(0) to v mod 2w. Then, iteratively, sets x(i) to (i + f * (x(i-1) xor (x(i-1) rshift w-2))) mod 2^w for i = 1 .. n-1. x(n) is the first value to be returned by operator().

3.
```
template<typename SeeqSeq> void seed(SeeqSeq & seq);
```

   Seeds a `mersenne_twister_engine` using values produced by seq.generate().

4.
```
template<typename It> void seed(It & first, It last);
```

   Sets the state of the generator using values from an iterator range.

5.
```
result_type operator()();
```

   Produces the next value of the generator.

6.
```
template<typename Iter> void generate(Iter first, Iter last);
```

   Fills a range with random values

7.
```
void discard(boost::uintmax_t z);
```

   Advances the state of the generator by `z` steps. Equivalent to

```
for(unsigned long long i = 0; i < z; ++i) {
    gen();
}
```

`mersenne_twister_engine` public static functions

1.
```
static result_type min();
```

   Returns the smallest value that the generator can produce.

2.
```
static result_type max();
```

   Returns the largest value that the generator can produce.

`mersenne_twister_engine` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const mersenne_twister_engine & mt);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Writes a `mersenne_twister_engine` to a `std::ostream`

2.

```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             mersenne_twister_engine & mt);
```

Reads a `mersenne_twister_engine` from a `std::istream`

3.

```
friend bool operator==(const mersenne_twister_engine & x_,
                       const mersenne_twister_engine & y_);
```

Returns true if the two generators are in the same state, and will thus produce identical sequences.

4.

```
friend bool operator!=(const mersenne_twister_engine & x_,
                       const mersenne_twister_engine & y_);
```

Returns true if the two generators are in different states.

Type definition mt11213b

mt11213b

Synopsis

```
// In header: <boost/random/mersenne_twister.hpp>


typedef mersenne_twister_en↵
gine< uint32_t, 32, 351, 175, 19, 0xccab8ee7, 11, 0xffffffff, 7, 0x31b6ab00, 15, 0xffe50000, 17, 1812433253 > mt11213b;
```

Description

The specializations mt11213b and mt19937 are from

> "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator", Makoto Matsumoto and Takuji Nishimura, ACM Transactions on Modeling and Computer Simulation: Special Issue on Uniform Random Number Generation, Vol. 8, No. 1, January 1998, pp. 3-30.

Type definition mt19937

mt19937

Synopsis

```
// In header: <boost/random/mersenne_twister.hpp>


typedef mersenne_twister_en↵
gine< uint32_t, 32, 624, 397, 31, 0x9908b0df, 11, 0xffffffff, 7, 0x9d2c5680, 15, 0xefc60000, 18, 1812433253 > mt19937;
```

Description

The specializations mt11213b and mt19937 are from

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

"Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator", Makoto Matsumoto and Takuji Nishimura, ACM Transactions on Modeling and Computer Simulation: Special Issue on Uniform Random Number Generation, Vol. 8, No. 1, January 1998, pp. 3-30.

Header <boost/random/negative_binomial_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename IntType = int, typename RealType = double>
      class negative_binomial_distribution;
  }
}
```

Class template negative_binomial_distribution

boost::random::negative_binomial_distribution

Synopsis

```
// In header: <boost/random/negative_binomial_distribution.hpp>

template<typename IntType = int, typename RealType = double>
class negative_binomial_distribution {
public:
  // types
  typedef IntType  result_type;
  typedef RealType input_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef negative_binomial_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(IntType = 1, RealType = 0.5);

    // public member functions
    IntType k() const;
    RealType p() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit negative_binomial_distribution(IntType = 1, RealType = 0.5);
  explicit negative_binomial_distribution(const param_type &);

  // public member functions
  template<typename URNG> IntType operator()(URNG &) const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  template<typename URNG> IntType operator()(URNG &, const param_type &) const;
  IntType k() const;
  RealType p() const;
  IntType min() const;
  IntType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();

  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, 
               const negative_binomial_distribution &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
    operator>>(std::basic_istream< CharT, Traits > &, 
               negative_binomial_distribution &);
  friend bool operator==(const negative_binomial_distribution &, 
                         const negative_binomial_distribution &);
  friend bool operator!=(const negative_binomial_distribution &, 
                         const negative_binomial_distribution &);
};
```

Description

The negative binomial distribution is an integer valued distribution with two parameters, `k` and `p`. The distribution produces non-negative values.

The distribution function is $P(i) = \binom{k+i-1}{i} p^k (1-p)^i$.

This implementation uses a gamma-poisson mixture.

`negative_binomial_distribution` public construct/copy/destruct

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

   Construct a `negative_binomial_distribution` object. `k` and `p` are the parameters of the distribution.

   Requires: k >=0 && 0 <= p <= 1

2.
```
explicit negative_binomial_distribution(const param_type & param);
```

   Construct an `negative_binomial_distribution` object from the parameters.

`negative_binomial_distribution` public member functions

1.
```
template<typename URNG> IntType operator()(URNG & urng) const;
```

   Returns a random variate distributed according to the negative binomial distribution.

2.
```
template<typename URNG> 
  IntType operator()(URNG & urng, const param_type & param) const;
```

   Returns a random variate distributed according to the negative binomial distribution with parameters specified by `param`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

3.
```
IntType k() const;
```

   Returns the `k` parameter of the distribution.

4.
```
RealType p() const;
```

   Returns the `p` parameter of the distribution.

5.
```
IntType min() const;
```

   Returns the smallest value that the distribution can produce.

6.
```
IntType max() const;
```

   Returns the largest value that the distribution can produce.

7.
```
param_type param() const;
```

   Returns the parameters of the distribution.

8.
```
void param(const param_type & param);
```

   Sets parameters of the distribution.

9.
```
void reset();
```

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

`negative_binomial_distribution` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const negative_binomial_distribution & bd);
```

   Writes the parameters of the distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             negative_binomial_distribution & bd);
```

   Reads the parameters of the distribution from a `std::istream`.

3.
```
friend bool operator==(const negative_binomial_distribution & lhs, 
                       const negative_binomial_distribution & rhs);
```

   Returns true if the two distributions will produce the same sequence of values, given equal generators.

4.
```
friend bool operator!=(const negative_binomial_distribution & lhs, 
                       const negative_binomial_distribution & rhs);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns true if the two distributions could produce different sequences of values, given equal generators.

Class param_type

boost::random::negative_binomial_distribution::param_type

Synopsis

```
// In header: <boost/random/negative_binomial_distribution.hpp>



class param_type {
public:
  // types
  typedef negative_binomial_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(IntType = 1, RealType = 0.5);

  // public member functions
  IntType k() const;
  RealType p() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.

```
explicit param_type(IntType k = 1, RealType p = 0.5);
```

Construct a `param_type` object. `k` and `p` are the parameters of the distribution.

Requires: k >=0 && 0 <= p <= 1

`param_type` public member functions

1.

```
IntType k() const;
```

Returns the `k` parameter of the distribution.

2.

```
RealType p() const;
```

Returns the `p` parameter of the distribution.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`param_type` friend functions

1.
   ```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const param_type & param);
```

   Writes the parameters of the distribution to a `std::ostream`.

2.
   ```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, param_type & param);
```

   Reads the parameters of the distribution from a `std::istream`.

3.
   ```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

   Returns true if the parameters have the same values.

4.
   ```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

   Returns true if the parameters have different values.

Header <boost/random/normal_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class normal_distribution;
  }
}
```

Class template normal_distribution

boost::random::normal_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/normal_distribution.hpp>

template<typename RealType = double>
class normal_distribution {
public:
  // types
  typedef RealType input_type;
  typedef RealType result_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef normal_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 0.0, RealType = 1.0);

    // public member functions
    RealType mean() const;
    RealType sigma() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit normal_distribution(const RealType & = 0.0, const RealType & = 1.0);
  explicit normal_distribution(const param_type &);

  // public member functions
  RealType mean() const;
  RealType sigma() const;
  RealType min() const;
  RealType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();
  template<typename Engine> result_type operator()(Engine &);
  template<typename URNG> result_type operator()(URNG &, const param_type &);

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const normal_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
                   const normal_distribution &);
  friend bool operator==(const normal_distribution &,
                         const normal_distribution &);
  friend bool operator!=(const normal_distribution &,
                         const normal_distribution &);
};
```

Description

Instantiations of class template normal_distribution model a random distribution . Such a distribution produces random numbers x distributed with probability density function $p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$, where mean and sigma are the parameters of the distribution.

`normal_distribution` public construct/copy/destruct

1.
   ```
   explicit normal_distribution(const RealType & mean = 0.0,
                                const RealType & sigma = 1.0);
   ```

   Constructs a `normal_distribution` object. `mean` and `sigma` are the parameters for the distribution.

   Requires: sigma >= 0

2.
   ```
   explicit normal_distribution(const param_type & param);
   ```

   Constructs a `normal_distribution` object from its parameters.

`normal_distribution` public member functions

1.
   ```
   RealType mean() const;
   ```

   Returns the mean of the distribution.

2.
   ```
   RealType sigma() const;
   ```

   Returns the standard deviation of the distribution.

3.
   ```
   RealType min() const;
   ```

   Returns the smallest value that the distribution can produce.

4.
   ```
   RealType max() const;
   ```

   Returns the largest value that the distribution can produce.

5.
   ```
   param_type param() const;
   ```

   Returns the parameters of the distribution.

6.
   ```
   void param(const param_type & param);
   ```

   Sets the parameters of the distribution.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

7.
```
void reset();
```

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

8.
```
template<typename Engine> result_type operator()(Engine & eng);
```

Returns a normal variate.

9.
```
template<typename URNG> 
  result_type operator()(URNG & urng, const param_type & param);
```

Returns a normal variate with parameters specified by `param`.

`normal_distribution` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const normal_distribution & nd);
```

Writes a `normal_distribution` to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const normal_distribution & nd);
```

Reads a `normal_distribution` from a `std::istream`.

3.
```
friend bool operator==(const normal_distribution & lhs, 
                       const normal_distribution & rhs);
```

Returns true if the two instances of `normal_distribution` will return identical sequences of values given equal generators.

4.
```
friend bool operator!=(const normal_distribution & lhs, 
                       const normal_distribution & rhs);
```

Returns true if the two instances of `normal_distribution` will return different sequences of values given equal generators.

Class param_type

boost::random::normal_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/normal_distribution.hpp>



class param_type {
public:
  // types
  typedef normal_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 0.0, RealType = 1.0);

  // public member functions
  RealType mean() const;
  RealType sigma() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.

```
explicit param_type(RealType mean = 0.0, RealType sigma = 1.0);
```

Constructs a `param_type` with a given mean and standard deviation.

Requires: sigma >= 0

`param_type` public member functions

1.

```
RealType mean() const;
```

Returns the mean of the distribution.

2.

```
RealType sigma() const;
```

Returns the standard deviation of the distribution.

`param_type` friend functions

1.

```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

Writes a `param_type` to a `std::ostream`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
                  const param_type & param);
```

Reads a `param_type` from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are the same.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are the different.

Header <boost/random/piecewise_constant_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double, typename WeightType = double>
      class piecewise_constant_distribution;
  }
}
```

Class template piecewise_constant_distribution

boost::random::piecewise_constant_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/piecewise_constant_distribution.hpp>

template<typename RealType = double, typename WeightType = double> 
class piecewise_constant_distribution {
public:
  // types
  typedef std::size_t input_type; 
  typedef RealType    result_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef piecewise_constant_distribution distribution_type;

    // construct/copy/destruct
    param_type();
    template<typename IntervalIter, typename WeightIter> 
      param_type(IntervalIter, IntervalIter, WeightIter);
    template<typename T, typename F> 
      param_type(const std::initializer_list< T > &, F);
    template<typename IntervalRange, typename WeightRange> 
      param_type(const IntervalRange &, const WeightRange &);
    template<typename F> param_type(std::size_t, RealType, RealType, F);

    // friend functions
    template<typename CharT, typename Traits> 
      friend std::basic_ostream< CharT, Traits > & 
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits> 
      friend std::basic_istream< CharT, Traits > & 
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);

    // public member functions
    std::vector< RealType > intervals() const;
    std::vector< RealType > densities() const;
  };

  // construct/copy/destruct
  piecewise_constant_distribution();
  template<typename IntervalIter, typename WeightIter> 
    piecewise_constant_distribution(IntervalIter, IntervalIter, WeightIter);
  template<typename T, typename F> 
    piecewise_constant_distribution(std::initializer_list< T >, F);
  template<typename IntervalsRange, typename WeightsRange> 
    piecewise_constant_distribution(const IntervalsRange &, 
                                    const WeightsRange &);
  template<typename F> 
    piecewise_constant_distribution(std::size_t, RealType, RealType, F);
  explicit piecewise_constant_distribution(const param_type &);

  // public member functions
  template<typename URNG> RealType operator()(URNG &) const;
  template<typename URNG> 
    RealType operator()(URNG &, const param_type &) const;
  result_type min() const;
  result_type max() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  std::vector< RealType > densities() const;
  std::vector< RealType > intervals() const;
  param_type param() const;
  void param(const param_type &);
  void reset();

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const piecewise_constant_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
               const piecewise_constant_distribution &);
  friend bool operator==(const piecewise_constant_distribution &,
                         const piecewise_constant_distribution &);
  friend bool operator!=(const piecewise_constant_distribution &,
                         const piecewise_constant_distribution &);
};
```

Description

The class `piecewise_constant_distribution` models a random distribution .

`piecewise_constant_distribution` public construct/copy/destruct

1.

```
piecewise_constant_distribution();
```

Creates a new `piecewise_constant_distribution` with a single interval, [0, 1).

2.

```
template<typename IntervalIter, typename WeightIter>
  piecewise_constant_distribution(IntervalIter first_interval,
                                  IntervalIter last_interval,
                                  WeightIter first_weight);
```

Constructs a `piecewise_constant_distribution` from two iterator ranges containing the interval boundaries and the interval weights. If there are less than two boundaries, then this is equivalent to the default constructor and creates a single interval, [0, 1).

The values of the interval boundaries must be strictly increasing, and the number of weights must be one less than the number of interval boundaries. If there are extra weights, they are ignored.

For example,

```
double intervals[] = { 0.0, 1.0, 4.0 };
double weights[] = { 1.0, 1.0 };
piecewise_constant_distribution<> dist(
    &intervals[0], &intervals[0] + 3, &weights[0]);
```

The distribution has a 50% chance of producing a value between 0 and 1 and a 50% chance of producing a value between 1 and 4.

3.

```
template<typename T, typename F>
  piecewise_constant_distribution(std::initializer_list< T > il, F f);
```

Constructs a `piecewise_constant_distribution` from an initializer_list containing the interval boundaries and a unary function specifying the weights. Each weight is determined by calling the function at the midpoint of the corresponding interval.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

If the initializer_list contains less than two elements, this is equivalent to the default constructor and the distribution will produce values uniformly distributed in the range [0, 1).

4.
```
template<typename IntervalsRange, typename WeightsRange> 
  piecewise_constant_distribution(const IntervalsRange & intervals_arg, 
                                  const WeightsRange & weights_arg);
```

Constructs a `piecewise_constant_distribution` from Boost.Range ranges holding the interval boundaries and the weights. If there are less than two interval boundaries, this is equivalent to the default constructor and the distribution will produce values uniformly distributed in the range [0, 1). The number of weights must be one less than the number of interval boundaries.

5.
```
template<typename F> 
  piecewise_constant_distribution(std::size_t nw, RealType xmin, 
                                  RealType xmax, F f);
```

Constructs a `piecewise_constant_distribution` that approximates a function. The range of the distribution is [xmin, xmax). This range is divided into nw equally sized intervals and the weights are found by calling the unary function f on the midpoints of the intervals.

6.
```
explicit piecewise_constant_distribution(const param_type & param);
```

Constructs a `piecewise_constant_distribution` from its parameters.

`piecewise_constant_distribution` public member functions

1.
```
template<typename URNG> RealType operator()(URNG & urng) const;
```

Returns a value distributed according to the parameters of the piecewist_constant_distribution.

2.
```
template<typename URNG> 
  RealType operator()(URNG & urng, const param_type & param) const;
```

Returns a value distributed according to the parameters specified by param.

3.
```
result_type min() const;
```

Returns the smallest value that the distribution can produce.

4.
```
result_type max() const;
```

Returns the largest value that the distribution can produce.

5.
```
std::vector< RealType > densities() const;
```

Returns a vector containing the probability density over each interval.

6.
```
std::vector< RealType > intervals() const;
```

Returns a vector containing the interval boundaries.

7.
```
param_type param() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns the parameters of the distribution.

8.
```
void param(const param_type & param);
```

Sets the parameters of the distribution.

9.
```
void reset();
```

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

`piecewise_constant_distribution` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const piecewise_constant_distribution & pcd);
```

Writes a distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const piecewise_constant_distribution & pcd);
```

Reads a distribution from a `std::istream`

3.
```
friend bool operator==(const piecewise_constant_distribution & lhs, 
                       const piecewise_constant_distribution & rhs);
```

Returns true if the two distributions will return the same sequence of values, when passed equal generators.

4.
```
friend bool operator!=(const piecewise_constant_distribution & lhs, 
                       const piecewise_constant_distribution & rhs);
```

Returns true if the two distributions may return different sequences of values, when passed equal generators.

Class param_type

boost::random::piecewise_constant_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/piecewise_constant_distribution.hpp>



class param_type {
public:
  // types
  typedef piecewise_constant_distribution distribution_type;

  // construct/copy/destruct
  param_type();
  template<typename IntervalIter, typename WeightIter>
    param_type(IntervalIter, IntervalIter, WeightIter);
  template<typename T, typename F>
    param_type(const std::initializer_list< T > &, F);
  template<typename IntervalRange, typename WeightRange>
    param_type(const IntervalRange &, const WeightRange &);
  template<typename F> param_type(std::size_t, RealType, RealType, F);

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);

  // public member functions
  std::vector< RealType > intervals() const;
  std::vector< RealType > densities() const;
};
```

Description

`param_type` public construct/copy/destruct

1.
   ```
   param_type();
   ```

   Constructs a `param_type` object, representing a distribution that produces values uniformly distributed in the range [0, 1).

2.
   ```
   template<typename IntervalIter, typename WeightIter>
     param_type(IntervalIter intervals_first, IntervalIter intervals_last,
                WeightIter weight_first);
   ```

   Constructs a `param_type` object from two iterator ranges containing the interval boundaries and the interval weights. If there are less than two boundaries, then this is equivalent to the default constructor and creates a single interval, [0, 1).

   The values of the interval boundaries must be strictly increasing, and the number of weights must be one less than the number of interval boundaries. If there are extra weights, they are ignored.

3.
   ```
   template<typename T, typename F>
     param_type(const std::initializer_list< T > & il, F f);
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Constructs a `param_type` object from an initializer_list containing the interval boundaries and a unary function specifying the weights. Each weight is determined by calling the function at the midpoint of the corresponding interval.

If the initializer_list contains less than two elements, this is equivalent to the default constructor and the distribution will produce values uniformly distributed in the range [0, 1).

4.
```
template<typename IntervalRange, typename WeightRange> 
  param_type(const IntervalRange & intervals_arg, 
             const WeightRange & weights_arg);
```

Constructs a `param_type` object from Boost.Range ranges holding the interval boundaries and the weights. If there are less than two interval boundaries, this is equivalent to the default constructor and the distribution will produce values uniformly distributed in the range [0, 1). The number of weights must be one less than the number of interval boundaries.

5.
```
template<typename F> 
  param_type(std::size_t nw, RealType xmin, RealType xmax, F f);
```

Constructs the parameters for a distribution that approximates a function. The range of the distribution is [xmin, xmax). This range is divided into nw equally sized intervals and the weights are found by calling the unary function f on the midpoints of the intervals.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const param_type & param);
```

Writes the parameters to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const param_type & param);
```

Reads the parameters from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are the same.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are different.

`param_type` public member functions

1.
```
std::vector< RealType > intervals() const;
```

Returns a vector containing the interval boundaries.

2.
```
std::vector< RealType > densities() const;
```

Returns a vector containing the probability densities over all the intervals of the distribution.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/random/piecewise_linear_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class piecewise_linear_distribution;
  }
}
```

Class template piecewise_linear_distribution

boost::random::piecewise_linear_distribution

Synopsis

```
// In header: <boost/random/piecewise_linear_distribution.hpp>

template<typename RealType = double> 
class piecewise_linear_distribution {
public:
  // types
  typedef std::size_t input_type; 
  typedef RealType    result_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef piecewise_linear_distribution distribution_type;

    // construct/copy/destruct
    param_type();
    template<typename IntervalIter, typename WeightIter> 
      param_type(IntervalIter, IntervalIter, WeightIter);
    template<typename T, typename F> 
      param_type(const std::initializer_list< T > &, F);
    template<typename IntervalRange, typename WeightRange> 
      param_type(const IntervalRange &, const WeightRange &);
    template<typename F> param_type(std::size_t, RealType, RealType, F);

    // friend functions
    template<typename CharT, typename Traits> 
      friend std::basic_ostream< CharT, Traits > & 
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits> 
      friend std::basic_istream< CharT, Traits > & 
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);

    // public member functions
    std::vector< RealType > intervals() const;
    std::vector< RealType > densities() const;
  };

  // construct/copy/destruct
  piecewise_linear_distribution();
  template<typename IntervalIter, typename WeightIter> 
    piecewise_linear_distribution(IntervalIter, IntervalIter, WeightIter);
  template<typename T, typename F>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    piecewise_linear_distribution(std::initializer_list< T >, F);
  template<typename IntervalsRange, typename WeightsRange> 
    piecewise_linear_distribution(const IntervalsRange &, 
                                  const WeightsRange &);
  template<typename F> 
    piecewise_linear_distribution(std::size_t, RealType, RealType, F);
  explicit piecewise_linear_distribution(const param_type &);

  // public member functions
  template<typename URNG> RealType operator()(URNG &) const;
  template<typename URNG> 
    RealType operator()(URNG &, const param_type &) const;
  result_type min() const;
  result_type max() const;
  std::vector< RealType > densities() const;
  std::vector< RealType > intervals() const;
  param_type param() const;
  void param(const param_type &);
  void reset();

  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, 
               const piecewise_linear_distribution &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
    operator>>(std::basic_istream< CharT, Traits > &, 
               const piecewise_linear_distribution &);
  friend bool operator==(const piecewise_linear_distribution &, 
                         const piecewise_linear_distribution &);
  friend bool operator!=(const piecewise_linear_distribution &, 
                         const piecewise_linear_distribution &);
};
```

Description

The class `piecewise_linear_distribution` models a random distribution .

`piecewise_linear_distribution` public construct/copy/destruct

1.
```
piecewise_linear_distribution();
```

   Creates a new `piecewise_linear_distribution` that produces values uniformly distributed in the range [0, 1).

2.
```
template<typename IntervalIter, typename WeightIter> 
  piecewise_linear_distribution(IntervalIter first_interval, 
                                IntervalIter last_interval, 
                                WeightIter first_weight);
```

   Constructs a `piecewise_linear_distribution` from two iterator ranges containing the interval boundaries and the weights at the boundaries. If there are fewer than two boundaries, then this is equivalent to the default constructor and creates a distribution that produces values uniformly distributed in the range [0, 1).

   The values of the interval boundaries must be strictly increasing, and the number of weights must be equal to the number of interval boundaries. If there are extra weights, they are ignored.

   For example,

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
double intervals[] = { 0.0, 1.0, 2.0 };
double weights[] = { 0.0, 1.0, 0.0 };
piecewise_constant_distribution<> dist(
    &intervals[0], &intervals[0] + 3, &weights[0]);
```

produces a triangle distribution.

3.
```
template<typename T, typename F>
  piecewise_linear_distribution(std::initializer_list< T > il, F f);
```

Constructs a `piecewise_linear_distribution` from an initializer_list containing the interval boundaries and a unary function specifying the weights. Each weight is determined by calling the function at the corresponding interval boundary.

If the initializer_list contains fewer than two elements, this is equivalent to the default constructor and the distribution will produce values uniformly distributed in the range [0, 1).

4.
```
template<typename IntervalsRange, typename WeightsRange>
  piecewise_linear_distribution(const IntervalsRange & intervals_arg,
                                const WeightsRange & weights_arg);
```

Constructs a `piecewise_linear_distribution` from Boost.Range ranges holding the interval boundaries and the weights. If there are fewer than two interval boundaries, this is equivalent to the default constructor and the distribution will produce values uniformly distributed in the range [0, 1). The number of weights must be equal to the number of interval boundaries.

5.
```
template<typename F>
  piecewise_linear_distribution(std::size_t nw, RealType xmin, RealType xmax,
                                F f);
```

Constructs a `piecewise_linear_distribution` that approximates a function. The range of the distribution is [xmin, xmax). This range is divided into nw equally sized intervals and the weights are found by calling the unary function f on the interval boundaries.

6.
```
explicit piecewise_linear_distribution(const param_type & param);
```

Constructs a `piecewise_linear_distribution` from its parameters.

`piecewise_linear_distribution` public member functions

1.
```
template<typename URNG> RealType operator()(URNG & urng) const;
```

Returns a value distributed according to the parameters of the `piecewise_linear_distribution`.

2.
```
template<typename URNG>
  RealType operator()(URNG & urng, const param_type & param) const;
```

Returns a value distributed according to the parameters specified by param.

3.
```
result_type min() const;
```

Returns the smallest value that the distribution can produce.

4.
```
result_type max() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns the largest value that the distribution can produce.

5.
```
std::vector< RealType > densities() const;
```

Returns a vector containing the probability densities at the interval boundaries.

6.
```
std::vector< RealType > intervals() const;
```

Returns a vector containing the interval boundaries.

7.
```
param_type param() const;
```

Returns the parameters of the distribution.

8.
```
void param(const param_type & param);
```

Sets the parameters of the distribution.

9.
```
void reset();
```

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

`piecewise_linear_distribution` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const piecewise_linear_distribution & pld);
```

Writes a distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const piecewise_linear_distribution & pld);
```

Reads a distribution from a `std::istream`

3.
```
friend bool operator==(const piecewise_linear_distribution & lhs,
                       const piecewise_linear_distribution & rhs);
```

Returns true if the two distributions will return the same sequence of values, when passed equal generators.

4.
```
friend bool operator!=(const piecewise_linear_distribution & lhs,
                       const piecewise_linear_distribution & rhs);
```

Returns true if the two distributions may return different sequences of values, when passed equal generators.

Class param_type

boost::random::piecewise_linear_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/piecewise_linear_distribution.hpp>



class param_type {
public:
  // types
  typedef piecewise_linear_distribution distribution_type;

  // construct/copy/destruct
  param_type();
  template<typename IntervalIter, typename WeightIter> 
    param_type(IntervalIter, IntervalIter, WeightIter);
  template<typename T, typename F> 
    param_type(const std::initializer_list< T > &, F);
  template<typename IntervalRange, typename WeightRange> 
    param_type(const IntervalRange &, const WeightRange &);
  template<typename F> param_type(std::size_t, RealType, RealType, F);

  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);

  // public member functions
  std::vector< RealType > intervals() const;
  std::vector< RealType > densities() const;
};
```

Description

`param_type` public construct/copy/destruct

1.

```
param_type();
```

Constructs a `param_type` object, representing a distribution that produces values uniformly distributed in the range [0, 1).

2.

```
template<typename IntervalIter, typename WeightIter> 
  param_type(IntervalIter intervals_first, IntervalIter intervals_last, 
             WeightIter weight_first);
```

Constructs a `param_type` object from two iterator ranges containing the interval boundaries and weights at the boundaries. If there are fewer than two boundaries, then this is equivalent to the default constructor and the distribution will produce values uniformly distributed in the range [0, 1).

The values of the interval boundaries must be strictly increasing, and the number of weights must be the same as the number of interval boundaries. If there are extra weights, they are ignored.

3.

```
template<typename T, typename F> 
  param_type(const std::initializer_list< T > & il, F f);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Constructs a `param_type` object from an initializer_list containing the interval boundaries and a unary function specifying the weights at the boundaries. Each weight is determined by calling the function at the corresponding point.

If the initializer_list contains fewer than two elements, this is equivalent to the default constructor and the distribution will produce values uniformly distributed in the range [0, 1).

4.
```
template<typename IntervalRange, typename WeightRange>
  param_type(const IntervalRange & intervals_arg,
             const WeightRange & weights_arg);
```

Constructs a `param_type` object from Boost.Range ranges holding the interval boundaries and the weights at the boundaries. If there are fewer than two interval boundaries, this is equivalent to the default constructor and the distribution will produce values uniformly distributed in the range [0, 1). The number of weights must be equal to the number of interval boundaries.

5.
```
template<typename F>
  param_type(std::size_t nw, RealType xmin, RealType xmax, F f);
```

Constructs the parameters for a distribution that approximates a function. The range of the distribution is [xmin, xmax). This range is divided into nw equally sized intervals and the weights are found by calling the unary function f on the boundaries of the intervals.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

Writes the parameters to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const param_type & param);
```

Reads the parameters from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are the same.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are different.

`param_type` public member functions

1.
```
std::vector< RealType > intervals() const;
```

Returns a vector containing the interval boundaries.

2.
```
std::vector< RealType > densities() const;
```

Returns a vector containing the probability densities at all the interval boundaries.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/random/poisson_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename IntType = int, typename RealType = double> 
      class poisson_distribution;
  }
}
```

Class template poisson_distribution

boost::random::poisson_distribution

Synopsis

```
// In header: <boost/random/poisson_distribution.hpp>

template<typename IntType = int, typename RealType = double> 
class poisson_distribution {
public:
  // types
  typedef IntType  result_type;
  typedef RealType input_type; 

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef poisson_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 1);

    // public member functions
    RealType mean() const;

    // friend functions
    template<typename CharT, typename Traits> 
      friend std::basic_ostream< CharT, Traits > & 
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits> 
      friend std::basic_istream< CharT, Traits > & 
      operator>>(std::basic_istream< CharT, Traits > &, param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit poisson_distribution(RealType = 1);
  explicit poisson_distribution(const param_type &);

  // public member functions
  template<typename URNG> IntType operator()(URNG &) const;
  template<typename URNG> IntType operator()(URNG &, const param_type &) const;
  RealType mean() const;
  IntType min() const;
  IntType max() const;
  param_type param() const;
  void param(const param_type &);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  void reset();

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const poisson_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, poisson_distribution &);
  friend bool operator==(const poisson_distribution &,
                         const poisson_distribution &);
  friend bool operator!=(const poisson_distribution &,
                         const poisson_distribution &);
};
```

Description

An instantiation of the class template `poisson_distribution` is a model of random distribution . The poisson distribution has $p(i) = \frac{e^{-\lambda}\lambda^i}{i!}$

This implementation is based on the PTRD algorithm described

> "The transformed rejection method for generating Poisson random variables", Wolfgang Hormann, Insurance: Mathematics and Economics Volume 12, Issue 1, February 1993, Pages 39-45

`poisson_distribution` public construct/copy/destruct

1.
```
explicit poisson_distribution(RealType mean = 1);
```

   Constructs a `poisson_distribution` with the parameter `mean`.

   Requires: mean > 0

2.
```
explicit poisson_distribution(const param_type & param);
```

   Construct an `poisson_distribution` object from the parameters.

`poisson_distribution` public member functions

1.
```
template<typename URNG> IntType operator()(URNG & urng) const;
```

   Returns a random variate distributed according to the poisson distribution.

2.
```
template<typename URNG>
  IntType operator()(URNG & urng, const param_type & param) const;
```

   Returns a random variate distributed according to the poisson distribution with parameters specified by param.

3.
```
RealType mean() const;
```

   Returns the "mean" parameter of the distribution.

4.
```
IntType min() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns the smallest value that the distribution can produce.

5.
```
IntType max() const;
```

Returns the largest value that the distribution can produce.

6.
```
param_type param() const;
```

Returns the parameters of the distribution.

7.
```
void param(const param_type & param);
```

Sets parameters of the distribution.

8.
```
void reset();
```

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

`poisson_distribution` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const poisson_distribution & pd);
```

Writes the parameters of the distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             poisson_distribution & pd);
```

Reads the parameters of the distribution from a `std::istream`.

3.
```
friend bool operator==(const poisson_distribution & lhs, 
                       const poisson_distribution & rhs);
```

Returns true if the two distributions will produce the same sequence of values, given equal generators.

4.
```
friend bool operator!=(const poisson_distribution & lhs, 
                       const poisson_distribution & rhs);
```

Returns true if the two distributions could produce different sequences of values, given equal generators.

Class param_type

boost::random::poisson_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/poisson_distribution.hpp>



class param_type {
public:
  // types
  typedef poisson_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 1);

  // public member functions
  RealType mean() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
   ```
   explicit param_type(RealType mean = 1);
   ```

   Construct a `param_type` object with the parameter "mean"

   Requires: mean > 0

`param_type` public member functions

1.
   ```
   RealType mean() const;
   ```

`param_type` friend functions

1.
   ```
   template<typename CharT, typename Traits>
     friend std::basic_ostream< CharT, Traits > &
     operator<<(std::basic_ostream< CharT, Traits > & os,
                const param_type & param);
   ```

   Writes the parameters of the distribution to a `std::ostream`.

2.
   ```
   template<typename CharT, typename Traits>
     friend std::basic_istream< CharT, Traits > &
     operator>>(std::basic_istream< CharT, Traits > & is, param_type & param);
   ```

   Reads the parameters of the distribution from a `std::istream`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

   Returns true if the parameters have the same values.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

   Returns true if the parameters have different values.

Header <boost/random/random_device.hpp>

```
namespace boost {
  namespace random {
    class random_device;
  }
}
```

Class random_device

boost::random::random_device

Synopsis

```
// In header: <boost/random/random_device.hpp>


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

  // construct/copy/destruct
  random_device();
  explicit random_device(const std::string &);
  ~random_device();

  // public static functions
  static result_type min();
  static result_type max();

  // public member functions
  double entropy() const;
  unsigned int operator()();
  template<typename Iter> void generate(Iter, Iter);

  // public data members
  static const bool has_fixed_range;
};
```

Description

Class random_device models a non-deterministic random number generator . It uses one or more implementation-defined stochastic processes to generate a sequence of uniformly distributed non-deterministic random numbers. For those environments where a non-deterministic random number generator is not available, class random_device must not be implemented. See

> "Randomness Recommendations for Security", D. Eastlake, S. Crocker, J. Schiller, Network Working Group, RFC 1750, December 1994

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

for further discussions.

> **Note**
>
> Some operating systems abstract the computer hardware enough to make it difficult to non-intrusively monitor stochastic processes. However, several do provide a special device for exactly this purpose. It seems to be impossible to emulate the functionality using Standard C++ only, so users should be aware that this class may not be available on all platforms.

Implementation Note for Linux

On the Linux operating system, token is interpreted as a filesystem path. It is assumed that this path denotes an operating system pseudo-device which generates a stream of non-deterministic random numbers. The pseudo-device should never signal an error or end-of-file. Otherwise, `std::ios_base::failure` is thrown. By default, random_device uses the /dev/urandom pseudo-device to retrieve the random numbers. Another option would be to specify the /dev/random pseudo-device, which blocks on reads if the entropy pool has no more random bits available.

Implementation Note for Windows

On the Windows operating system, token is interpreted as the name of a cryptographic service provider. By default random_device uses MS_DEF_PROV.

Performance

The test program nondet_random_speed.cpp measures the execution times of the random_device.hpp implementation of the above algorithms in a tight loop. The performance has been evaluated on an Intel(R) Core(TM) i7 CPU Q 840 @ 1.87GHz, 1867 Mhz with Visual C++ 2010, Microsoft Windows 7 Professional and with gcc 4.4.5, Ubuntu Linux 2.6.35-25-generic.

Platform	time per invocation [microseconds]
Windows	2.9
Linux	1.7

The measurement error is estimated at +/- 1 usec.

`random_device` public construct/copy/destruct

1.
   ```
   random_device();
   ```

   Constructs a `random_device`, optionally using the default device.

2.
   ```
   explicit random_device(const std::string & token);
   ```

   Constructs a `random_device`, optionally using the given token as an access specification (for example, a URL) to some implementation-defined service for monitoring a stochastic process.

3.
   ```
   ~random_device();
   ```

`random_device` public static functions

1.
   ```
   static result_type min();
   ```

   Returns the smallest value that the `random_device` can produce.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
static result_type max();
```

Returns the largest value that the `random_device` can produce.

`random_device` public member functions

1.
```
double entropy() const;
```

Returns: An entropy estimate for the random numbers returned by operator(), in the range min() to log2(max()+1). A deterministic random number generator (e.g. a pseudo-random number engine) has entropy 0.

Throws: Nothing.

2.
```
unsigned int operator()();
```

Returns a random value in the range [min, max].

3.
```
template<typename Iter> void generate(Iter begin, Iter end);
```

Fills a range with random 32-bit values.

Header <boost/random/random_number_generator.hpp>

```
namespace boost {
  namespace random {
    template<typename URNG, typename IntType = long>
      class random_number_generator;
  }
}
```

Class template random_number_generator

boost::random::random_number_generator

Synopsis

```
// In header: <boost/random/random_number_generator.hpp>

template<typename URNG, typename IntType = long>
class random_number_generator {
public:
  // types
  typedef URNG    base_type;
  typedef IntType argument_type;
  typedef IntType result_type;

  // construct/copy/destruct
  random_number_generator(base_type &);

  // public member functions
  result_type operator()(argument_type);
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Instantiations of class template random_number_generator model a RandomNumberGenerator (std:25.2.11 [lib.alg.random.shuffle]). On each invocation, it returns a uniformly distributed integer in the range [0..n).

The template parameter IntType shall denote some integer-like value type.

`random_number_generator` public construct/copy/destruct

1.
```
random_number_generator(base_type & rng);
```

   Constructs a `random_number_generator` functor with the given uniform random number generator as the underlying source of random numbers.

`random_number_generator` public member functions

1.
```
result_type operator()(argument_type n);
```

   Returns a value in the range [0, n)

Header <boost/random/ranlux.hpp>

```
namespace boost {
  namespace random {
    typedef subtract_with_carry_engine< uint32_t, 24, 10, 24 > ranlux_base;
    typedef subtract_with_carry_01_engine< float, 24, 10, 24 > ranlux_base_01;
    typedef subtract_with_carry_01_engine< double, 48, 10, 24 > ranlux64_base_01;
    typedef discard_block_engine< ranlux_base, 223, 24 > ranlux3;
    typedef discard_block_engine< ranlux_base, 389, 24 > ranlux4;
    typedef discard_block_engine< ranlux_base_01, 223, 24 > ranlux3_01;
    typedef discard_block_engine< ranlux_base_01, 389, 24 > ranlux4_01;
    typedef discard_block_engine< ranlux64_base_01, 223, 24 > ranlux64_3_01;
    typedef discard_block_engine< ranlux64_base_01, 389, 24 > ranlux64_4_01;
    typedef subtract_with_carry_engine< uint64_t, 48, 10, 24 > ranlux64_base;
    typedef discard_block_engine< ranlux64_base, 223, 24 > ranlux64_3;
    typedef discard_block_engine< ranlux64_base, 389, 24 > ranlux64_4;
    typedef subtract_with_carry_engine< uint32_t, 24, 10, 24 > ranlux24_base;
    typedef subtract_with_carry_engine< uint64_t, 48, 5, 12 > ranlux48_base;
    typedef discard_block_engine< ranlux24_base, 223, 23 > ranlux24;
    typedef discard_block_engine< ranlux48_base, 389, 11 > ranlux48;
  }
}
```

Type definition ranlux3

ranlux3

Synopsis

```
// In header: <boost/random/ranlux.hpp>


typedef discard_block_engine< ranlux_base, 223, 24 > ranlux3;
```

Description

The ranlux family of generators are described in

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

> "A portable high-quality random number generator for lattice field theory calculations", M. Luescher, Computer Physics Communications, 79 (1994) pp 100-110.

The levels are given in

> "RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Luescher", F. James, Computer Physics Communications 79 (1994) 111-114

Type definition ranlux4

ranlux4

Synopsis

```
// In header: <boost/random/ranlux.hpp>


typedef discard_block_engine< ranlux_base, 389, 24 > ranlux4;
```

Description

The ranlux family of generators are described in

> "A portable high-quality random number generator for lattice field theory calculations", M. Luescher, Computer Physics Communications, 79 (1994) pp 100-110.

The levels are given in

> "RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Luescher", F. James, Computer Physics Communications 79 (1994) 111-114

Type definition ranlux3_01

ranlux3_01

Synopsis

```
// In header: <boost/random/ranlux.hpp>


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

Description

The ranlux family of generators are described in

> "A portable high-quality random number generator for lattice field theory calculations", M. Luescher, Computer Physics Communications, 79 (1994) pp 100-110.

The levels are given in

> "RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Luescher", F. James, Computer Physics Communications 79 (1994) 111-114

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Type definition ranlux4_01

ranlux4_01

Synopsis

```
// In header: <boost/random/ranlux.hpp>


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

Description

The ranlux family of generators are described in

> "A portable high-quality random number generator for lattice field theory calculations", M. Luescher, Computer Physics Communications, 79 (1994) pp 100-110.

The levels are given in

> "RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Luescher", F. James, Computer Physics Communications 79 (1994) 111-114

Type definition ranlux64_3_01

ranlux64_3_01

Synopsis

```
// In header: <boost/random/ranlux.hpp>


typedef discard_block_engine< ranlux64_base_01, 223, 24 > ranlux64_3_01;
```

Description

The ranlux family of generators are described in

> "A portable high-quality random number generator for lattice field theory calculations", M. Luescher, Computer Physics Communications, 79 (1994) pp 100-110.

The levels are given in

> "RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Luescher", F. James, Computer Physics Communications 79 (1994) 111-114

Type definition ranlux64_4_01

ranlux64_4_01

Synopsis

```
// In header: <boost/random/ranlux.hpp>


typedef discard_block_engine< ranlux64_base_01, 389, 24 > ranlux64_4_01;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

The ranlux family of generators are described in

> "A portable high-quality random number generator for lattice field theory calculations", M. Luescher, Computer Physics Communications, 79 (1994) pp 100-110.

The levels are given in

> "RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Luescher", F. James, Computer Physics Communications 79 (1994) 111-114

Type definition ranlux64_3

ranlux64_3

Synopsis

```
// In header: <boost/random/ranlux.hpp>


typedef discard_block_engine< ranlux64_base, 223, 24 > ranlux64_3;
```

Description

The ranlux family of generators are described in

> "A portable high-quality random number generator for lattice field theory calculations", M. Luescher, Computer Physics Communications, 79 (1994) pp 100-110.

The levels are given in

> "RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Luescher", F. James, Computer Physics Communications 79 (1994) 111-114

Type definition ranlux64_4

ranlux64_4

Synopsis

```
// In header: <boost/random/ranlux.hpp>


typedef discard_block_engine< ranlux64_base, 389, 24 > ranlux64_4;
```

Description

The ranlux family of generators are described in

> "A portable high-quality random number generator for lattice field theory calculations", M. Luescher, Computer Physics Communications, 79 (1994) pp 100-110.

The levels are given in

> "RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Luescher", F. James, Computer Physics Communications 79 (1994) 111-114

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Header <boost/random/seed_seq.hpp>

```
namespace boost {
  namespace random {
    class seed_seq;
  }
}
```

Class seed_seq

boost::random::seed_seq

Synopsis

```
// In header: <boost/random/seed_seq.hpp>


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

  // construct/copy/destruct
  seed_seq();
  template<typename T> seed_seq(const std::initializer_list< T > &);
  template<typename Iter> seed_seq(Iter, Iter);
  template<typename Range> explicit seed_seq(const Range &);

  // public member functions
  template<typename Iter> void generate(Iter, Iter) const;
  std::size_t size() const;
  template<typename Iter> void param(Iter);
};
```

Description

The class `seed_seq` stores a sequence of 32-bit words for seeding a pseudo-random number generator . These words will be combined to fill the entire state of the generator.

seed_seq public construct/copy/destruct

1.
```
seed_seq();
```

   Initializes a `seed_seq` to hold an empty sequence.

2.
```
template<typename T> seed_seq(const std::initializer_list< T > & il);
```

   Initializes the sequence from an initializer_list.

3.
```
template<typename Iter> seed_seq(Iter first, Iter last);
```

   Initializes the sequence from an iterator range.

4.
```
template<typename Range> explicit seed_seq(const Range & range);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Initializes the sequence from Boost.Range range.

seed_seq public member functions

1.
```
template<typename Iter> void generate(Iter first, Iter last) const;
```

Fills a range with 32-bit values based on the stored sequence.

Requires: Iter must be a Random Access Iterator whose value type is an unsigned integral type at least 32 bits wide.

2.
```
std::size_t size() const;
```

Returns the size of the sequence.

3.
```
template<typename Iter> void param(Iter out);
```

Writes the stored sequence to iter.

Header <boost/random/shuffle_order.hpp>

```
namespace boost {
  namespace random {
    template<typename UniformRandomNumberGenerator, std::size_t k> 
      class shuffle_order_engine;
    typedef shuffle_order_engine< linear_congruential_en↵
gine< uint32_t, 1366, 150889, 714025 >, 97 > kreutzer1986;
    typedef shuffle_order_engine< minstd_rand0, 256 > knuth_b;
  }
}
```

Class template shuffle_order_engine

boost::random::shuffle_order_engine

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/shuffle_order.hpp>

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

  // construct/copy/destruct
  shuffle_order_engine();
  explicit shuffle_order_engine(result_type);
  template<typename SeedSeq> explicit shuffle_order_engine(SeedSeq &);
  explicit shuffle_order_engine(const base_type &);
  explicit shuffle_order_engine(base_type &&);
  template<typename It> shuffle_order_engine(It &, It);

  // public member functions
  void seed();
  void seed(result_type);
  template<typename SeedSeq> void seed(SeedSeq &);
  template<typename It> void seed(It &, It);
  const base_type & base() const;
  result_type operator()();
  void discard(boost::uintmax_t);
  template<typename Iter> void generate(Iter, Iter);

  // public static functions
  static result_type min();
  static result_type max();

  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, 
               const shuffle_order_engine &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
    operator>>(std::basic_istream< CharT, Traits > &, 
               const shuffle_order_engine &);
  friend bool operator==(const shuffle_order_engine &, 
                         const shuffle_order_engine &);
  friend bool operator!=(const shuffle_order_engine &, 
                         const shuffle_order_engine &);

  // public data members
  static const bool has_fixed_range;
  static const std::size_t buffer_size;
  static const std::size_t table_size;
};
```

Description

Instatiations of class template `shuffle_order_engine` model a pseudo-random number generator . It mixes the output of some (usually linear_congruential_engine) uniform random number generator to get better statistical properties. The algorithm is described in

> "Improving a poor random number generator", Carter Bays and S.D. Durham, ACM Transactions on Mathematical Software, Vol 2, No. 1, March 1976, pp. 59-64. http://doi.acm.org/10.1145/355666.355670

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

The output of the base generator is buffered in an array of length k. Every output X(n) has a second role: It gives an index into the array where X(n+1) will be retrieved. Used array elements are replaced with fresh output from the base generator.

Template parameters are the base generator and the array length k, which should be around 100.

`shuffle_order_engine` public construct/copy/destruct

1.
   ```
   shuffle_order_engine();
   ```

   Constructs a `shuffle_order_engine` by invoking the default constructor of the base generator.

   Complexity: Exactly k+1 invocations of the base generator.

2.
   ```
   explicit shuffle_order_engine(result_type s);
   ```

   Constructs a `shuffle_output_engine` by invoking the one-argument constructor of the base generator with the parameter seed.

   Complexity: Exactly k+1 invocations of the base generator.

3.
   ```
   template<typename SeedSeq> explicit shuffle_order_engine(SeedSeq & seq);
   ```

4.
   ```
   explicit shuffle_order_engine(const base_type & rng);
   ```

   Constructs a `shuffle_output_engine` by using a copy of the provided generator.

   Precondition: The template argument UniformRandomNumberGenerator shall denote a CopyConstructible type.

   Complexity: Exactly k+1 invocations of the base generator.

5.
   ```
   explicit shuffle_order_engine(base_type && rng);
   ```

6.
   ```
   template<typename It> shuffle_order_engine(It & first, It last);
   ```

`shuffle_order_engine` public member functions

1.
   ```
   void seed();
   ```

2.
   ```
   void seed(result_type seed);
   ```

   Invokes the one-argument seed method of the base generator with the parameter seed and re-initializes the internal buffer array.

   Complexity: Exactly k+1 invocations of the base generator.

3.
   ```
   template<typename SeedSeq> void seed(SeedSeq & seq);
   ```

   Invokes the one-argument seed method of the base generator with the parameter seq and re-initializes the internal buffer array.

   Complexity: Exactly k+1 invocations of the base generator.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

4.
```
template<typename It> void seed(It & first, It last);
```

5.
```
const base_type & base() const;
```

6.
```
result_type operator()();
```

7.
```
void discard(boost::uintmax_t z);
```

Advances the generator by z steps.

8.
```
template<typename Iter> void generate(Iter first, Iter last);
```

Fills a range with pseudo-random values.

`shuffle_order_engine` public static functions

1.
```
static result_type min();
```

Returns the smallest value that the generator can produce.

2.
```
static result_type max();
```

Returns the largest value that the generator can produce.

`shuffle_order_engine` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const shuffle_order_engine & s);
```

Writes a `shuffle_order_engine` to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const shuffle_order_engine & s);
```

Reads a `shuffle_order_engine` from a `std::istream`.

3.
```
friend bool operator==(const shuffle_order_engine & x, 
                       const shuffle_order_engine & y);
```

Returns true if the two generators will produce identical sequences.

4.
```
friend bool operator!=(const shuffle_order_engine & lhs, 
                       const shuffle_order_engine & rhs);
```

Returns true if the two generators will produce different sequences.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Type definition kreutzer1986

kreutzer1986

Synopsis

```
// In header: <boost/random/shuffle_order.hpp>


typedef shuffle_order_engine< linear_congruential_en↵
gine< uint32_t, 1366, 150889, 714025 >, 97 > kreutzer1986;
```

Description

According to Harry Erwin (private e-mail), the specialization `kreutzer1986` was suggested in:

> "System Simulation: Programming Styles and Languages (International Computer Science Series)", Wolfgang Kreutzer, Addison-Wesley, December 1986.

Type definition knuth_b

knuth_b

Synopsis

```
// In header: <boost/random/shuffle_order.hpp>


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

Description

The specialization `knuth_b` is specified by the C++ standard. It is described in

> "The Art of Computer Programming, Second Edition, Volume 2, Seminumerical Algorithms", Donald Knuth, Addison-Wesley, 1981.

Header <boost/random/student_t_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class student_t_distribution;
  }
}
```

Class template student_t_distribution

boost::random::student_t_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/student_t_distribution.hpp>

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

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef student_t_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 1.0);

    // public member functions
    RealType n() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit student_t_distribution(RealType = 1.0);
  explicit student_t_distribution(const param_type &);

  // public member functions
  template<typename URNG> RealType operator()(URNG &);
  template<typename URNG>
    RealType operator()(URNG &, const param_type &) const;
  RealType n() const;
  RealType min() const;
  RealType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const student_t_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
               const student_t_distribution &);
  friend bool operator==(const student_t_distribution &,
                         const student_t_distribution &);
  friend bool operator!=(const student_t_distribution &,
                         const student_t_distribution &);
};
```

Description

The Student t distribution is a real valued distribution with one parameter n, the number of degrees of freedom.

It has $p(x) = \frac{1}{\sqrt{n\pi}} \frac{\Gamma((n+1)/2)}{\Gamma(n/2)} \left(1+\frac{x^2}{n}\right)^{-(n+1)/2}$.

`student_t_distribution` public construct/copy/destruct

1.
```
explicit student_t_distribution(RealType n = 1.0);
```

   Constructs an `student_t_distribution` with "n" degrees of freedom.

   Requires: n > 0

2.
```
explicit student_t_distribution(const param_type & param);
```

   Constructs an `student_t_distribution` from its parameters.

`student_t_distribution` public member functions

1.
```
template<typename URNG> RealType operator()(URNG & urng);
```

   Returns a random variate distributed according to the Student t distribution.

2.
```
template<typename URNG>
  RealType operator()(URNG & urng, const param_type & param) const;
```

   Returns a random variate distributed accordint to the Student t distribution with parameters specified by `param`.

3.
```
RealType n() const;
```

   Returns the number of degrees of freedom.

4.
```
RealType min() const;
```

   Returns the smallest value that the distribution can produce.

5.
```
RealType max() const;
```

   Returns the largest value that the distribution can produce.

6.
```
param_type param() const;
```

   Returns the parameters of the distribution.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

7.
```
void param(const param_type & param);
```

Sets the parameters of the distribution.

8.
```
void reset();
```

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

`student_t_distribution` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const student_t_distribution & td);
```

Writes a `student_t_distribution` to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const student_t_distribution & td);
```

Reads a `student_t_distribution` from a `std::istream`.

3.
```
friend bool operator==(const student_t_distribution & lhs, 
                       const student_t_distribution & rhs);
```

Returns true if the two instances of `student_t_distribution` will return identical sequences of values given equal generators.

4.
```
friend bool operator!=(const student_t_distribution & lhs, 
                       const student_t_distribution & rhs);
```

Returns true if the two instances of `student_t_distribution` will return different sequences of values given equal generators.

Class param_type

boost::random::student_t_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/student_t_distribution.hpp>



class param_type {
public:
  // types
  typedef student_t_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 1.0);

  // public member functions
  RealType n() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
```
explicit param_type(RealType n = 1.0);
```

   Constructs a `param_type` with "n" degrees of freedom.

   Requires: n > 0

`param_type` public member functions

1.
```
RealType n() const;
```

   Returns the number of degrees of freedom of the distribution.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

   Writes a `param_type` to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const param_type & param);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Reads a `param_type` from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are the same.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are the different.

Header <boost/random/subtract_with_carry.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType, std::size_t w, std::size_t s, std::size_t r>
      class subtract_with_carry_01_engine;
    template<typename IntType, std::size_t w, std::size_t s, std::size_t r>
      class subtract_with_carry_engine;
  }
}
```

Class template subtract_with_carry_01_engine

boost::random::subtract_with_carry_01_engine

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/subtract_with_carry.hpp>

template<typename RealType, std::size_t w, std::size_t s, std::size_t r>
class subtract_with_carry_01_engine {
public:
  // types
  typedef RealType result_type;

  // construct/copy/destruct
  subtract_with_carry_01_engine();
  explicit subtract_with_carry_01_engine(boost::uint32_t);
  template<typename SeedSeq> explicit subtract_with_carry_01_engine(SeedSeq &);
  template<typename It> subtract_with_carry_01_engine(It &, It);

  // public member functions
  void seed();
  void seed(boost::uint32_t);
  template<typename SeedSeq> void seed(SeedSeq &);
  template<typename It> void seed(It &, It);
  result_type operator()();
  void discard(boost::uintmax_t);
  template<typename Iter> void generate(Iter, Iter);

  // public static functions
  static result_type min();
  static result_type max();

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const subtract_with_carry_01_engine &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
               const subtract_with_carry_01_engine &);
  friend bool operator==(const subtract_with_carry_01_engine &,
                         const subtract_with_carry_01_engine &);
  friend bool operator!=(const subtract_with_carry_01_engine &,
                         const subtract_with_carry_01_engine &);

  // public data members
  static const bool has_fixed_range;
  static const std::size_t word_size;
  static const std::size_t long_lag;
  static const std::size_t short_lag;
  static const boost::uint32_t default_seed;
};
```

Description

Instantiations of subtract_with_carry_01_engine model a pseudo-random number generator . The algorithm is described in

> "A New Class of Random Number Generators", George Marsaglia and Arif Zaman, Annals of Applied Probability, Volume 1, Number 3 (1991), 462-480.

`subtract_with_carry_01_engine` public construct/copy/destruct

1.
```
subtract_with_carry_01_engine();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Creates a new `subtract_with_carry_01_engine` using the default seed.

2.
```
explicit subtract_with_carry_01_engine(boost::uint32_t value);
```

Creates a new `subtract_with_carry_01_engine` and seeds it with value.

3.
```
template<typename SeedSeq>
  explicit subtract_with_carry_01_engine(SeedSeq & seq);
```

Creates a new `subtract_with_carry_01_engine` and seeds with values produced by seq.generate().

4.
```
template<typename It> subtract_with_carry_01_engine(It & first, It last);
```

Creates a new `subtract_with_carry_01_engine` and seeds it with values from a range. Advances first to point one past the last consumed value. If the range does not contain enough elements to fill the entire state, throws `std::invalid_argument`.

`subtract_with_carry_01_engine` public member functions

1.
```
void seed();
```

Seeds the generator with the default seed.

2.
```
void seed(boost::uint32_t value);
```

Seeds the generator with `value`.

3.
```
template<typename SeedSeq> void seed(SeedSeq & seq);
```

Seeds the generator with values produced by `seq.generate()`.

4.
```
template<typename It> void seed(It & first, It last);
```

Seeds the generator with values from a range. Updates first to point one past the last consumed element. If there are not enough elements in the range to fill the entire state, throws `std::invalid_argument`.

5.
```
result_type operator()();
```

Returns the next value of the generator.

6.
```
void discard(boost::uintmax_t z);
```

Advances the state of the generator by `z`.

7.
```
template<typename Iter> void generate(Iter first, Iter last);
```

Fills a range with random values.

`subtract_with_carry_01_engine` public static functions

1.
```
static result_type min();
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns the smallest value that the generator can produce.

2.
```
static result_type max();
```

Returns the largest value that the generator can produce.

`subtract_with_carry_01_engine` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const subtract_with_carry_01_engine & f);
```

Writes a `subtract_with_carry_01_engine` to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const subtract_with_carry_01_engine & f);
```

Reads a `subtract_with_carry_01_engine` from a `std::istream`.

3.
```
friend bool operator==(const subtract_with_carry_01_engine & x,
                       const subtract_with_carry_01_engine & y);
```

Returns true if the two generators will produce identical sequences.

4.
```
friend bool operator!=(const subtract_with_carry_01_engine & lhs,
                       const subtract_with_carry_01_engine & rhs);
```

Returns true if the two generators will produce different sequences.

Class template subtract_with_carry_engine

boost::random::subtract_with_carry_engine

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/subtract_with_carry.hpp>

template<typename IntType, std::size_t w, std::size_t s, std::size_t r>
class subtract_with_carry_engine {
public:
  // types
  typedef IntType result_type;

  // construct/copy/destruct
  subtract_with_carry_engine();
  explicit subtract_with_carry_engine(IntType);
  template<typename SeedSeq> explicit subtract_with_carry_engine(SeedSeq &);
  template<typename It> subtract_with_carry_engine(It &, It);

  // public member functions
  void seed();
  void seed(IntType);
  template<typename SeedSeq> void seed(SeedSeq &);
  template<typename It> void seed(It &, It);
  result_type operator()();
  void discard(boost::uintmax_t);
  template<typename It> void generate(It, It);

  // public static functions
  static result_type min();
  static result_type max();

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const subtract_with_carry_engine &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
               const subtract_with_carry_engine &);
  friend bool operator==(const subtract_with_carry_engine &,
                         const subtract_with_carry_engine &);
  friend bool operator!=(const subtract_with_carry_engine &,
                         const subtract_with_carry_engine &);

  // public data members
  static const std::size_t word_size;
  static const std::size_t long_lag;
  static const std::size_t short_lag;
  static const uint32_t default_seed;
  static const bool has_fixed_range;
  static const result_type modulus;
};
```

Description

Instantiations of `subtract_with_carry_engine` model a pseudo-random number generator . The algorithm is described in

> "A New Class of Random Number Generators", George Marsaglia and Arif Zaman, Annals of Applied Probability, Volume 1, Number 3 (1991), 462-480.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`subtract_with_carry_engine` public construct/copy/destruct

1.
```
subtract_with_carry_engine();
```

   Constructs a new `subtract_with_carry_engine` and seeds it with the default seed.

2.
```
explicit subtract_with_carry_engine(IntType value);
```

   Constructs a new `subtract_with_carry_engine` and seeds it with `value`.

3.
```
template<typename SeedSeq> explicit subtract_with_carry_engine(SeedSeq & seq);
```

   Constructs a new `subtract_with_carry_engine` and seeds it with values produced by `seq.generate()`.

4.
```
template<typename It> subtract_with_carry_engine(It & first, It last);
```

   Constructs a new `subtract_with_carry_engine` and seeds it with values from a range. first is updated to point one past the last value consumed. If there are not enough elements in the range to fill the entire state of the generator, throws `std::invalid_argument`.

`subtract_with_carry_engine` public member functions

1.
```
void seed();
```

   Seeds the generator with the default seed.

2.
```
void seed(IntType value);
```

3.
```
template<typename SeedSeq> void seed(SeedSeq & seq);
```

   Seeds the generator with values produced by `seq.generate()`.

4.
```
template<typename It> void seed(It & first, It last);
```

   Seeds the generator with values from a range. Updates `first` to point one past the last consumed value. If the range does not contain enough elements to fill the entire state of the generator, throws `std::invalid_argument`.

5.
```
result_type operator()();
```

   Returns the next value of the generator.

6.
```
void discard(boost::uintmax_t z);
```

   Advances the state of the generator by `z`.

7.
```
template<typename It> void generate(It first, It last);
```

   Fills a range with random values.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`subtract_with_carry_engine` public static functions

1.
```
static result_type min();
```

   Returns the smallest value that the generator can produce.

2.
```
static result_type max();
```

   Returns the largest value that the generator can produce.

`subtract_with_carry_engine` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const subtract_with_carry_engine & f);
```

   Writes a `subtract_with_carry_engine` to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const subtract_with_carry_engine & f);
```

   Reads a `subtract_with_carry_engine` from a `std::istream`.

3.
```
friend bool operator==(const subtract_with_carry_engine & x,
                       const subtract_with_carry_engine & y);
```

   Returns true if the two generators will produce identical sequences of values.

4.
```
friend bool operator!=(const subtract_with_carry_engine & lhs,
                       const subtract_with_carry_engine & rhs);
```

   Returns true if the two generators will produce different sequences of values.

Header <boost/random/taus88.hpp>

```
namespace boost {
  namespace random {
    typedef xor_combine_engine< xor_combine_engine< linear_feedback_shift_en↵
gine< uint32_t, 32, 31, 13, 12 >, 0, linear_feedback_shift_en↵
gine< uint32_t, 32, 29, 2, 4 >, 0 >, 0, linear_feedback_shift_en↵
gine< uint32_t, 32, 28, 3, 17 >, 0 > taus88;
  }
}
```

Type definition taus88

taus88

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/taus88.hpp>


typedef xor_combine_engine< xor_combine_engine< linear_feedback_shift_en↵
gine< uint32_t, 32, 31, 13, 12 >, 0, linear_feedback_shift_en↵
gine< uint32_t, 32, 29, 2, 4 >, 0 >, 0, linear_feedback_shift_en↵
gine< uint32_t, 32, 28, 3, 17 >, 0 > taus88;
```

Description

The specialization taus88 was suggested in

> "Maximally Equidistributed Combined Tausworthe Generators", Pierre L'Ecuyer, Mathematics of Computation, Volume 65, Number 213, January 1996, Pages 203-213

Header <boost/random/triangle_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class triangle_distribution;
  }
}
```

Class template triangle_distribution

boost::random::triangle_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/triangle_distribution.hpp>

template<typename RealType = double>
class triangle_distribution {
public:
  // types
  typedef RealType input_type;
  typedef RealType result_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef triangle_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 0.0, RealType = 0.5, RealType = 1.0);

    // public member functions
    RealType a() const;
    RealType b() const;
    RealType c() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit triangle_distribution(RealType = 0.0, RealType = 0.5,
                                 RealType = 1.0);
  explicit triangle_distribution(const param_type &);

  // public member functions
  result_type a() const;
  result_type b() const;
  result_type c() const;
  RealType min() const;
  RealType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();
  template<typename Engine> result_type operator()(Engine &);
  template<typename Engine>
    result_type operator()(Engine &, const param_type &);

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const triangle_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    operator>>(std::basic_istream< CharT, Traits > &,
               const triangle_distribution &);
  friend bool operator==(const triangle_distribution &,
                         const triangle_distribution &);
  friend bool operator!=(const triangle_distribution &,
                         const triangle_distribution &);
};
```

Description

Instantiations of `triangle_distribution` model a random distribution . A `triangle_distribution` has three parameters, `a`, `b`, and `c`, which are the smallest, the most probable and the largest values of the distribution respectively.

`triangle_distribution` public construct/copy/destruct

1.
   ```
   explicit triangle_distribution(RealType a = 0.0, RealType b = 0.5,
                                  RealType c = 1.0);
   ```

   Constructs a `triangle_distribution` with the parameters `a`, `b`, and `c`.

   Preconditions: a <= b <= c.

2.
   ```
   explicit triangle_distribution(const param_type & param);
   ```

   Constructs a `triangle_distribution` from its parameters.

`triangle_distribution` public member functions

1.
   ```
   result_type a() const;
   ```

   Returns the `a` parameter of the distribution

2.
   ```
   result_type b() const;
   ```

   Returns the `b` parameter of the distribution

3.
   ```
   result_type c() const;
   ```

   Returns the `c` parameter of the distribution

4.
   ```
   RealType min() const;
   ```

   Returns the smallest value that the distribution can produce.

5.
   ```
   RealType max() const;
   ```

   Returns the largest value that the distribution can produce.

6.
   ```
   param_type param() const;
   ```

   Returns the parameters of the distribution.

7.
   ```
   void param(const param_type & param);
   ```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Sets the parameters of the distribution.

8.
```
void reset();
```

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

9.
```
template<typename Engine> result_type operator()(Engine & eng);
```

Returns a random variate distributed according to the triangle distribution.

10.
```
template<typename Engine>
  result_type operator()(Engine & eng, const param_type & param);
```

Returns a random variate distributed according to the triangle distribution with parameters specified by param.

`triangle_distribution` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const triangle_distribution & td);
```

Writes the distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const triangle_distribution & td);
```

Reads the distribution from a `std::istream`.

3.
```
friend bool operator==(const triangle_distribution & lhs,
                       const triangle_distribution & rhs);
```

Returns true if the two distributions will produce identical sequences of values given equal generators.

4.
```
friend bool operator!=(const triangle_distribution & lhs,
                       const triangle_distribution & rhs);
```

Returns true if the two distributions may produce different sequences of values given equal generators.

Class param_type

boost::random::triangle_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/triangle_distribution.hpp>



class param_type {
public:
  // types
  typedef triangle_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 0.0, RealType = 0.5, RealType = 1.0);

  // public member functions
  RealType a() const;
  RealType b() const;
  RealType c() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
```
explicit param_type(RealType a = 0.0, RealType b = 0.5, RealType c = 1.0);
```

   Constructs the parameters of a `triangle_distribution`.

`param_type` public member functions

1.
```
RealType a() const;
```

   Returns the minimum value of the distribution.

2.
```
RealType b() const;
```

   Returns the mode of the distribution.

3.
```
RealType c() const;
```

   Returns the maximum value of the distribution.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`param_type` friend functions

1.
   ```
   template<typename CharT, typename Traits> 
     friend std::basic_ostream< CharT, Traits > & 
     operator<<(std::basic_ostream< CharT, Traits > & os, 
                const param_type & param);
   ```

   Writes the parameters to a `std::ostream`.

2.
   ```
   template<typename CharT, typename Traits> 
     friend std::basic_istream< CharT, Traits > & 
     operator>>(std::basic_istream< CharT, Traits > & is, 
                const param_type & param);
   ```

   Reads the parameters from a `std::istream`.

3.
   ```
   friend bool operator==(const param_type & lhs, const param_type & rhs);
   ```

   Returns true if the two sets of parameters are equal.

4.
   ```
   friend bool operator!=(const param_type & lhs, const param_type & rhs);
   ```

   Returns true if the two sets of parameters are different.

Header <boost/random/uniform_01.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class uniform_01;
  }
}
```

Class template uniform_01

boost::random::uniform_01

Synopsis

```
// In header: <boost/random/uniform_01.hpp>

template<typename RealType = double> 
class uniform_01 {
public:
  // types
  typedef RealType input_type; 
  typedef RealType result_type;

  // public member functions
  result_type min() const;
  result_type max() const;
  void reset();
  template<typename Engine> result_type operator()(Engine &);
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

The distribution function uniform_01 models a random distribution . On each invocation, it returns a random floating-point value uniformly distributed in the range [0..1).

The template parameter RealType shall denote a float-like value type with support for binary operators +, -, and /.

Note: The current implementation is buggy, because it may not fill all of the mantissa with random bits. I'm unsure how to fill a (to-be-invented) `boost::bigfloat` class with random bits efficiently. It's probably time for a traits class.

`uniform_01` public member functions

1.
```
result_type min() const;
```

2.
```
result_type max() const;
```

3.
```
void reset();
```

4.
```
template<typename Engine> result_type operator()(Engine & eng);
```

Header <boost/random/uniform_int_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename IntType = int> class uniform_int_distribution;
  }
}
```

Class template uniform_int_distribution

boost::random::uniform_int_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/uniform_int_distribution.hpp>

template<typename IntType = int>
class uniform_int_distribution {
public:
  // types
  typedef IntType input_type;
  typedef IntType result_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef uniform_int_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(IntType = 0,
                        IntType = (std::numeric_limits< IntType >::max)());

    // public member functions
    IntType a() const;
    IntType b() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit uniform_int_distribution(IntType = 0,
                                    IntType = (std::numeric_limits< IntType >::max)());
  explicit uniform_int_distribution(const param_type &);

  // public member functions
  IntType min() const;
  IntType max() const;
  IntType a() const;
  IntType b() const;
  param_type param() const;
  void param(const param_type &);
  void reset();
  template<typename Engine> result_type operator()(Engine &) const;
  template<typename Engine>
    result_type operator()(Engine &, const param_type &) const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const uniform_int_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
                  const uniform_int_distribution &);
  friend bool operator==(const uniform_int_distribution &,
                         const uniform_int_distribution &);
  friend bool operator!=(const uniform_int_distribution &,
                         const uniform_int_distribution &);
};
```

Description

The class template uniform_int_distribution models a random distribution . On each invocation, it returns a random integer value uniformly distributed in the set of integers {min, min+1, min+2, ..., max}.

The template parameter IntType shall denote an integer-like value type.

`uniform_int_distribution` public construct/copy/destruct

1.
```
explicit uniform_int_distribution(IntType min = 0,
                                  IntType max = (std::numeric_limits< IntType >::max)());
```

   Constructs a `uniform_int_distribution`. `min` and `max` are the parameters of the distribution.

   Requires: min <= max

2.
```
explicit uniform_int_distribution(const param_type & param);
```

   Constructs a `uniform_int_distribution` from its parameters.

`uniform_int_distribution` public member functions

1.
```
IntType min() const;
```

   Returns the minimum value of the distribution

2.
```
IntType max() const;
```

   Returns the maximum value of the distribution

3.
```
IntType a() const;
```

   Returns the minimum value of the distribution

4.
```
IntType b() const;
```

   Returns the maximum value of the distribution

5.
```
param_type param() const;
```

   Returns the parameters of the distribution.

6.
```
void param(const param_type & param);
```

   Sets the parameters of the distribution.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

7.
```
void reset();
```

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

8.
```
template<typename Engine> result_type operator()(Engine & eng) const;
```

Returns an integer uniformly distributed in the range [min, max].

9.
```
template<typename Engine>
  result_type operator()(Engine & eng, const param_type & param) const;
```

Returns an integer uniformly distributed in the range [param.a(), param.b()].

`uniform_int_distribution` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const uniform_int_distribution & ud);
```

Writes the distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const uniform_int_distribution & ud);
```

Reads the distribution from a `std::istream`.

3.
```
friend bool operator==(const uniform_int_distribution & lhs,
                       const uniform_int_distribution & rhs);
```

Returns true if the two distributions will produce identical sequences of values given equal generators.

4.
```
friend bool operator!=(const uniform_int_distribution & lhs,
                       const uniform_int_distribution & rhs);
```

Returns true if the two distributions may produce different sequences of values given equal generators.

Class param_type

boost::random::uniform_int_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/uniform_int_distribution.hpp>



class param_type {
public:
  // types
  typedef uniform_int_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(IntType = 0,
                      IntType = (std::numeric_limits< IntType >::max)());

  // public member functions
  IntType a() const;
  IntType b() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
```
explicit param_type(IntType min = 0,
                    IntType max = (std::numeric_limits< IntType >::max)());
```

   Constructs the parameters of a `uniform_int_distribution`.

   Requires min <= max

`param_type` public member functions

1.
```
IntType a() const;
```

   Returns the minimum value of the distribution.

2.
```
IntType b() const;
```

   Returns the maximum value of the distribution.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Writes the parameters to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const param_type & param);
```

Reads the parameters from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are equal.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are different.

Header <boost/random/uniform_on_sphere.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double,
             typename Cont = std::vector<RealType> >
      class uniform_on_sphere;
  }
}
```

Class template uniform_on_sphere

boost::random::uniform_on_sphere

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/uniform_on_sphere.hpp>

template<typename RealType = double, typename Cont = std::vector<RealType> > 
class uniform_on_sphere {
public:
  // types
  typedef RealType input_type; 
  typedef Cont     result_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef uniform_on_sphere distribution_type;

    // construct/copy/destruct
    explicit param_type(int = 2);

    // public member functions
    int dim() const;

    // friend functions
    template<typename CharT, typename Traits> 
      friend std::basic_ostream< CharT, Traits > & 
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits> 
      friend std::basic_istream< CharT, Traits > & 
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit uniform_on_sphere(int = 2);
  explicit uniform_on_sphere(const param_type &);

  // public member functions
  int dim() const;
  param_type param() const;
  void param(const param_type &);
  result_type min() const;
  result_type max() const;
  void reset();
  template<typename Engine> const result_type & operator()(Engine &);
  template<typename Engine> 
    result_type operator()(Engine &, const param_type &) const;

  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, 
               const uniform_on_sphere &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
    operator>>(std::basic_istream< CharT, Traits > &, 
               const uniform_on_sphere &);
  friend bool operator==(const uniform_on_sphere &, const uniform_on_sphere &);
  friend bool operator!=(const uniform_on_sphere &, const uniform_on_sphere &);
};
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Description

Instantiations of class template uniform_on_sphere model a random distribution . Such a distribution produces random numbers uniformly distributed on the unit sphere of arbitrary dimension `dim`. The `Cont` template parameter must be a STL-like container type with begin and end operations returning non-const ForwardIterators of type `Cont::iterator`.

`uniform_on_sphere` public construct/copy/destruct

1.
```
explicit uniform_on_sphere(int dim = 2);
```

   Constructs a `uniform_on_sphere` distribution. `dim` is the dimension of the sphere.

   Requires: dim >= 0

2.
```
explicit uniform_on_sphere(const param_type & param);
```

   Constructs a `uniform_on_sphere` distribution from its parameters.

`uniform_on_sphere` public member functions

1.
```
int dim() const;
```

   Returns the dimension of the sphere.

2.
```
param_type param() const;
```

   Returns the parameters of the distribution.

3.
```
void param(const param_type & param);
```

   Sets the parameters of the distribution.

4.
```
result_type min() const;
```

   Returns the smallest value that the distribution can produce. Note that this is required to approximate the standard library's requirements. The behavior is defined according to lexicographical comparison so that for a container type of std::vector, dist.min() <= x <= dist.max() where x is any value produced by the distribution.

5.
```
result_type max() const;
```

   Returns the largest value that the distribution can produce. Note that this is required to approximate the standard library's requirements. The behavior is defined according to lexicographical comparison so that for a container type of std::vector, dist.min() <= x <= dist.max() where x is any value produced by the distribution.

6.
```
void reset();
```

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

7.
```
template<typename Engine> const result_type & operator()(Engine & eng);
```

   Returns a point uniformly distributed over the surface of a sphere of dimension dim().

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

8.
```
template<typename Engine>
  result_type operator()(Engine & eng, const param_type & param) const;
```

Returns a point uniformly distributed over the surface of a sphere of dimension param.dim().

`uniform_on_sphere` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const uniform_on_sphere & sd);
```

Writes the distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const uniform_on_sphere & sd);
```

Reads the distribution from a `std::istream`.

3.
```
friend bool operator==(const uniform_on_sphere & lhs,
                       const uniform_on_sphere & rhs);
```

Returns true if the two distributions will produce identical sequences of values, given equal generators.

4.
```
friend bool operator!=(const uniform_on_sphere & lhs,
                       const uniform_on_sphere & rhs);
```

Returns true if the two distributions may produce different sequences of values, given equal generators.

Class param_type

boost::random::uniform_on_sphere::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/uniform_on_sphere.hpp>



class param_type {
public:
  // types
  typedef uniform_on_sphere distribution_type;

  // construct/copy/destruct
  explicit param_type(int = 2);

  // public member functions
  int dim() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.

```
explicit param_type(int dim = 2);
```

Constructs the parameters of a `uniform_on_sphere` distribution, given the dimension of the sphere.

`param_type` public member functions

1.

```
int dim() const;
```

Returns the dimension of the sphere.

`param_type` friend functions

1.

```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

Writes the parameters to a `std::ostream`.

2.

```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const param_type & param);
```

Reads the parameters from a `std::istream`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

   Returns true if the two sets of parameters are equal.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

   Returns true if the two sets of parameters are different.

Header <boost/random/uniform_real_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class uniform_real_distribution;
  }
}
```

Class template uniform_real_distribution

boost::random::uniform_real_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/uniform_real_distribution.hpp>

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

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef uniform_real_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 0.0, RealType = 1.0);

    // public member functions
    RealType a() const;
    RealType b() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit uniform_real_distribution(RealType = 0.0, RealType = 1.0);
  explicit uniform_real_distribution(const param_type &);

  // public member functions
  RealType min() const;
  RealType max() const;
  RealType a() const;
  RealType b() const;
  param_type param() const;
  void param(const param_type &);
  void reset();
  template<typename Engine> result_type operator()(Engine &) const;
  template<typename Engine>
    result_type operator()(Engine &, const param_type &) const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const uniform_real_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
                const uniform_real_distribution &);
  friend bool operator==(const uniform_real_distribution &,
                         const uniform_real_distribution &);
  friend bool operator!=(const uniform_real_distribution &,
                         const uniform_real_distribution &);
};
```

Description

The class template uniform_real_distribution models a random distribution . On each invocation, it returns a random floating-point value uniformly distributed in the range [min..max).

`uniform_real_distribution` public construct/copy/destruct

1.
   ```
   explicit uniform_real_distribution(RealType min = 0.0, RealType max = 1.0);
   ```

   Constructs a `uniform_real_distribution`. `min` and `max` are the parameters of the distribution.

   Requires: min <= max

2.
   ```
   explicit uniform_real_distribution(const param_type & param);
   ```

   Constructs a `uniform_real_distribution` from its parameters.

`uniform_real_distribution` public member functions

1.
   ```
   RealType min() const;
   ```

   Returns the minimum value of the distribution

2.
   ```
   RealType max() const;
   ```

   Returns the maximum value of the distribution

3.
   ```
   RealType a() const;
   ```

   Returns the minimum value of the distribution

4.
   ```
   RealType b() const;
   ```

   Returns the maximum value of the distribution

5.
   ```
   param_type param() const;
   ```

   Returns the parameters of the distribution.

6.
   ```
   void param(const param_type & param);
   ```

   Sets the parameters of the distribution.

7.
   ```
   void reset();
   ```

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

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

8.

```
template<typename Engine> result_type operator()(Engine & eng) const;
```

Returns a value uniformly distributed in the range [min, max).

9.

```
template<typename Engine> 
  result_type operator()(Engine & eng, const param_type & param) const;
```

Returns a value uniformly distributed in the range [param.a(), param.b()).

`uniform_real_distribution` friend functions

1.

```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const uniform_real_distribution & ud);
```

Writes the distribution to a `std::ostream`.

2.

```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const uniform_real_distribution & ud);
```

Reads the distribution from a `std::istream`.

3.

```
friend bool operator==(const uniform_real_distribution & lhs, 
                       const uniform_real_distribution & rhs);
```

Returns true if the two distributions will produce identical sequences of values given equal generators.

4.

```
friend bool operator!=(const uniform_real_distribution & lhs, 
                       const uniform_real_distribution & rhs);
```

Returns true if the two distributions may produce different sequences of values given equal generators.

Class param_type

boost::random::uniform_real_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/uniform_real_distribution.hpp>



class param_type {
public:
  // types
  typedef uniform_real_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 0.0, RealType = 1.0);

  // public member functions
  RealType a() const;
  RealType b() const;

  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.

```
explicit param_type(RealType min = 0.0, RealType max = 1.0);
```

Constructs the parameters of a `uniform_real_distribution`.

Requires min <= max

`param_type` public member functions

1.

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

Returns the minimum value of the distribution.

2.

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

Returns the maximum value of the distribution.

`param_type` friend functions

1.

```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const param_type & param);
```

Writes the parameters to a `std::ostream`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const param_type & param);
```

Reads the parameters from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are equal.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are different.

Header <boost/random/uniform_smallint.hpp>

```
namespace boost {
  namespace random {
    template<typename IntType = int> class uniform_smallint;
  }
}
```

Class template uniform_smallint

boost::random::uniform_smallint

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/uniform_smallint.hpp>

template<typename IntType = int>
class uniform_smallint {
public:
  // types
  typedef IntType input_type;
  typedef IntType result_type;

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef uniform_smallint distribution_type;

    // construct/copy/destruct
    param_type(IntType = 0, IntType = 9);

    // public member functions
    IntType a() const;
    IntType b() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit uniform_smallint(IntType = 0, IntType = 9);
  explicit uniform_smallint(const param_type &);

  // public member functions
  result_type a() const;
  result_type b() const;
  result_type min() const;
  result_type max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();
  template<typename Engine> result_type operator()(Engine &) const;
  template<typename Engine>
    result_type operator()(Engine &, const param_type &) const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const uniform_smallint &);
  template<typename CharT, typename Traits>
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
               const uniform_smallint &);
  friend bool operator==(const uniform_smallint &, const uniform_smallint &);
  friend bool operator!=(const uniform_smallint &, const uniform_smallint &);
};
```

Description

The distribution function uniform_smallint models a random distribution . On each invocation, it returns a random integer value uniformly distributed in the set of integer numbers {min, min+1, min+2, ..., max}. It assumes that the desired range (max-min+1) is small compared to the range of the underlying source of random numbers and thus makes no attempt to limit quantization errors.

Let $r_{out} = (\max - \min + 1)$ the desired range of integer numbers, and let r_{base} be the range of the underlying source of random numbers. Then, for the uniform distribution, the theoretical probability for any number i in the range r_{out} will be $p_{out}(i) = \frac{1}{r_{out}}$. Likewise, assume a uniform distribution on r_{base} for the underlying source of random numbers, i.e. $p_{base}(i) = \frac{1}{r_{base}}$. Let $p_{out\_s}(i)$ denote the random distribution generated by `uniform_smallint`. Then the sum over all i in r_{out} of $\left(\frac{p_{out\_s}(i)}{p_{out}(i)} - 1\right)^2$ shall not exceed $\frac{r_{out}}{r_{base}^2}(r_{base} \bmod r_{out})(r_{out} - r_{base} \bmod r_{out})$.

The template parameter IntType shall denote an integer-like value type.

Note

The property above is the square sum of the relative differences in probabilities between the desired uniform distribution $p_{out}(i)$ and the generated distribution $p_{out\_s}(i)$. The property can be fulfilled with the calculation $(\mathrm{base\_rng} \bmod r_{out})$, as follows: Let $r = r_{base} \bmod r_{out}$. The base distribution on r_{base} is folded onto the range r_{out}. The numbers i < r have assigned $\left\lfloor \frac{r_{base}}{r_{out}} \right\rfloor + 1$ numbers of the base distribution, the rest has only $\left\lfloor \frac{r_{base}}{r_{out}} \right\rfloor$. Therefore, $p_{out\_s}(i) = \left(\left\lfloor \frac{r_{base}}{r_{out}} \right\rfloor + 1\right) / r_{base}$ for i < r and $p_{out\_s}(i) = \left\lfloor \frac{r_{base}}{r_{out}} \right\rfloor / r_{base}$ otherwise. Substituting this in the above sum formula leads to the desired result.

Note: The upper bound for $(r_{base} \bmod r_{out})(r_{out} - r_{base} \bmod r_{out})$ is $\frac{r_{out}^2}{4}$. Regarding the upper bound for the square sum of the relative quantization error of $\frac{r_{out}^3}{4 r_{base}^2}$, it seems wise to either choose r_{base} so that $r_{base} > 10 r_{out}^2$ or ensure that r_{base} is divisible by r_{out}.

`uniform_smallint` public construct/copy/destruct

1.
```
explicit uniform_smallint(IntType min = 0, IntType max = 9);
```

Constructs a `uniform_smallint`. `min` and `max` are the lower and upper bounds of the output range, respectively.

2.
```
explicit uniform_smallint(const param_type & param);
```

Constructs a `uniform_smallint` from its parameters.

`uniform_smallint` public member functions

1.
```
result_type a() const;
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns the minimum value of the distribution.

2.
```
result_type b() const;
```

Returns the maximum value of the distribution.

3.
```
result_type min() const;
```

Returns the minimum value of the distribution.

4.
```
result_type max() const;
```

Returns the maximum value of the distribution.

5.
```
param_type param() const;
```

Returns the parameters of the distribution.

6.
```
void param(const param_type & param);
```

Sets the parameters of the distribution.

7.
```
void reset();
```

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

8.
```
template<typename Engine> result_type operator()(Engine & eng) const;
```

Returns a value uniformly distributed in the range [min(), max()].

9.
```
template<typename Engine>
  result_type operator()(Engine & eng, const param_type & param) const;
```

Returns a value uniformly distributed in the range [param.a(), param.b()].

`uniform_smallint` friend functions

1.
```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const uniform_smallint & ud);
```

Writes the distribution to a `std::ostream`.

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const uniform_smallint & ud);
```

Reads the distribution from a `std::istream`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

3.
```
friend bool operator==(const uniform_smallint & lhs, 
                       const uniform_smallint & rhs);
```

Returns true if the two distributions will produce identical sequences of values given equal generators.

4.
```
friend bool operator!=(const uniform_smallint & lhs, 
                       const uniform_smallint & rhs);
```

Returns true if the two distributions may produce different sequences of values given equal generators.

Class param_type

boost::random::uniform_smallint::param_type

Synopsis

```
// In header: <boost/random/uniform_smallint.hpp>



class param_type {
public:
  // types
  typedef uniform_smallint distribution_type;

  // construct/copy/destruct
  param_type(IntType = 0, IntType = 9);

  // public member functions
  IntType a() const;
  IntType b() const;

  // friend functions
  template<typename CharT, typename Traits> 
    friend std::basic_ostream< CharT, Traits > & 
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits> 
    friend std::basic_istream< CharT, Traits > & 
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.
```
param_type(IntType min = 0, IntType max = 9);
```

constructs the parameters of a `uniform_smallint` distribution.

`param_type` public member functions

1.
```
IntType a() const;
```

Returns the minimum value.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
IntType b() const;
```

Returns the maximum value.

`param_type` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const param_type & param);
```

Writes the parameters to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const param_type & param);
```

Reads the parameters from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are equal.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

Returns true if the two sets of parameters are different.

Header <boost/random/variate_generator.hpp>

```
namespace boost {
  template<typename Engine, typename Distribution> class variate_generator;
}
```

Class template variate_generator

boost::variate_generator

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/variate_generator.hpp>

template<typename Engine, typename Distribution> 
class variate_generator {
public:
  // types
  typedef helper_type::value_type   engine_value_type; 
  typedef Engine                    engine_type;       
  typedef Distribution              distribution_type; 
  typedef Distribution::result_type result_type;       

  // construct/copy/destruct
  variate_generator(Engine, Distribution);

  // public member functions
  result_type operator()();
  template<typename T> result_type operator()(const T &);
  engine_value_type & engine();
  const engine_value_type & engine() const;
  distribution_type & distribution();
  const distribution_type & distribution() const;
  result_type min() const;
  result_type max() const;
};
```

Description

A random variate generator is used to join a random number generator together with a random number distribution. Boost.Random provides a vast choice of generators as well as distributions .

The argument for the template parameter Engine shall be of the form U, U&, or U*, where U models a uniform random number generator . Then, the member engine_value_type names U (not the pointer or reference to U).

Specializations of `variate_generator` satisfy the requirements of CopyConstructible. They also satisfy the requirements of Assignable unless the template parameter Engine is of the form U&.

The complexity of all functions specified in this section is constant. No function described in this section except the constructor throws an exception.

`variate_generator` public construct/copy/destruct

1.
```
variate_generator(Engine e, Distribution d);
```

Constructs a `variate_generator` object with the associated uniform random number generator eng and the associated random distribution d.

Throws: If and what the copy constructor of Engine or Distribution throws.

`variate_generator` public member functions

1.
```
result_type operator()();
```

Returns: distribution()(engine())

2.
```
template<typename T> result_type operator()(const T & value);
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Returns: distribution()(engine(), value).

3.
```
engine_value_type & engine();
```

Returns: A reference to the associated uniform random number generator.

4.
```
const engine_value_type & engine() const;
```

Returns: A reference to the associated uniform random number generator.

5.
```
distribution_type & distribution();
```

Returns: A reference to the associated random distribution .

6.
```
const distribution_type & distribution() const;
```

Returns: A reference to the associated random distribution.

7.
```
result_type min() const;
```

Precondition: distribution().min() is well-formed

Returns: distribution().min()

8.
```
result_type max() const;
```

Precondition: distribution().max() is well-formed

Returns: distribution().max()

Header <boost/random/weibull_distribution.hpp>

```
namespace boost {
  namespace random {
    template<typename RealType = double> class weibull_distribution;
  }
}
```

Class template weibull_distribution

boost::random::weibull_distribution

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/weibull_distribution.hpp>

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

  // member classes/structs/unions

  class param_type {
  public:
    // types
    typedef weibull_distribution distribution_type;

    // construct/copy/destruct
    explicit param_type(RealType = 1.0, RealType = 1.0);

    // public member functions
    RealType a() const;
    RealType b() const;

    // friend functions
    template<typename CharT, typename Traits>
      friend std::basic_ostream< CharT, Traits > &
      operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
    template<typename CharT, typename Traits>
      friend std::basic_istream< CharT, Traits > &
      operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
    friend bool operator==(const param_type &, const param_type &);
    friend bool operator!=(const param_type &, const param_type &);
  };

  // construct/copy/destruct
  explicit weibull_distribution(RealType = 1.0, RealType = 1.0);
  explicit weibull_distribution(const param_type &);

  // public member functions
  template<typename URNG> RealType operator()(URNG &) const;
  template<typename URNG>
    RealType operator()(URNG &, const param_type &) const;
  RealType a() const;
  RealType b() const;
  RealType min() const;
  RealType max() const;
  param_type param() const;
  void param(const param_type &);
  void reset();

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const weibull_distribution &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
```

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
               const weibull_distribution &);
  friend bool operator==(const weibull_distribution &,
                         const weibull_distribution &);
  friend bool operator!=(const weibull_distribution &,
                         const weibull_distribution &);
};
```

Description

The Weibull distribution is a real valued distribution with two parameters a and b, producing values >= 0.

It has $p(x) = \frac{a}{b}\left(\frac{x}{b}\right)^{a-1} e^{-\left(\frac{x}{b}\right)^a}$.

`weibull_distribution` public construct/copy/destruct

1.
```
explicit weibull_distribution(RealType a = 1.0, RealType b = 1.0);
```

   Constructs a `weibull_distribution` from its "a" and "b" parameters.

   Requires: a > 0 && b > 0

2.
```
explicit weibull_distribution(const param_type & param);
```

   Constructs a `weibull_distribution` from its parameters.

`weibull_distribution` public member functions

1.
```
template<typename URNG> RealType operator()(URNG & urng) const;
```

   Returns a random variate distributed according to the `weibull_distribution`.

2.
```
template<typename URNG>
  RealType operator()(URNG & urng, const param_type & param) const;
```

   Returns a random variate distributed accordint to the Weibull distribution with parameters specified by `param`.

3.
```
RealType a() const;
```

   Returns the "a" parameter of the distribution.

4.
```
RealType b() const;
```

   Returns the "b" parameter of the distribution.

5.
```
RealType min() const;
```

   Returns the smallest value that the distribution can produce.

6.
```
RealType max() const;
```

   Returns the largest value that the distribution can produce.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

7.

```
param_type param() const;
```

Returns the parameters of the distribution.

8.

```
void param(const param_type & param);
```

Sets the parameters of the distribution.

9.

```
void reset();
```

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

`weibull_distribution` friend functions

1.

```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const weibull_distribution & wd);
```

Writes a `weibull_distribution` to a `std::ostream`.

2.

```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const weibull_distribution & wd);
```

Reads a `weibull_distribution` from a `std::istream`.

3.

```
friend bool operator==(const weibull_distribution & lhs, 
                       const weibull_distribution & rhs);
```

Returns true if the two instances of `weibull_distribution` will return identical sequences of values given equal generators.

4.

```
friend bool operator!=(const weibull_distribution & lhs, 
                       const weibull_distribution & rhs);
```

Returns true if the two instances of `weibull_distribution` will return different sequences of values given equal generators.

Class param_type

boost::random::weibull_distribution::param_type

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/weibull_distribution.hpp>



class param_type {
public:
  // types
  typedef weibull_distribution distribution_type;

  // construct/copy/destruct
  explicit param_type(RealType = 1.0, RealType = 1.0);

  // public member functions
  RealType a() const;
  RealType b() const;

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &, const param_type &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &, const param_type &);
  friend bool operator==(const param_type &, const param_type &);
  friend bool operator!=(const param_type &, const param_type &);
};
```

Description

`param_type` public construct/copy/destruct

1.

```
explicit param_type(RealType a = 1.0, RealType b = 1.0);
```

Constructs a `param_type` from the "a" and "b" parameters of the distribution.

Requires: a > 0 && b > 0

`param_type` public member functions

1.

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

Returns the "a" parameter of the distribtuion.

2.

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

Returns the "b" parameter of the distribution.

`param_type` friend functions

1.

```
template<typename CharT, typename Traits>
  friend std::basic_ostream< CharT, Traits > &
  operator<<(std::basic_ostream< CharT, Traits > & os,
             const param_type & param);
```

Writes a `param_type` to a `std::ostream`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

2.
```
template<typename CharT, typename Traits>
  friend std::basic_istream< CharT, Traits > &
  operator>>(std::basic_istream< CharT, Traits > & is,
             const param_type & param);
```

   Reads a `param_type` from a `std::istream`.

3.
```
friend bool operator==(const param_type & lhs, const param_type & rhs);
```

   Returns true if the two sets of parameters are the same.

4.
```
friend bool operator!=(const param_type & lhs, const param_type & rhs);
```

   Returns true if the two sets of parameters are the different.

Header <boost/random/xor_combine.hpp>

```
namespace boost {
  namespace random {
    template<typename URNG1, int s1, typename URNG2, int s2>
      class xor_combine_engine;
  }
}
```

Class template xor_combine_engine

boost::random::xor_combine_engine

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Synopsis

```
// In header: <boost/random/xor_combine.hpp>

template<typename URNG1, int s1, typename URNG2, int s2>
class xor_combine_engine {
public:
  // types
  typedef URNG1                   base1_type;
  typedef URNG2                   base2_type;
  typedef base1_type::result_type result_type;

  // construct/copy/destruct
  xor_combine_engine();
  xor_combine_engine(const base1_type &, const base2_type &);
  explicit xor_combine_engine(result_type);
  template<typename SeedSeq> explicit xor_combine_engine(SeedSeq &);
  template<typename It> xor_combine_engine(It &, It);

  // public member functions
  void seed();
  void seed(result_type);
  template<typename SeedSeq> void seed(SeedSeq &);
  template<typename It> void seed(It &, It);
  const base1_type & base1() const;
  const base2_type & base2() const;
  result_type operator()();
  template<typename Iter> void generate(Iter, Iter);
  void discard(boost::uintmax_t);

  // public static functions
  static result_type min();
  static result_type max();

  // friend functions
  template<typename CharT, typename Traits>
    friend std::basic_ostream< CharT, Traits > &
    operator<<(std::basic_ostream< CharT, Traits > &,
               const xor_combine_engine &);
  template<typename CharT, typename Traits>
    friend std::basic_istream< CharT, Traits > &
    operator>>(std::basic_istream< CharT, Traits > &,
               const xor_combine_engine &);
  friend bool operator==(const xor_combine_engine &,
                         const xor_combine_engine &);
  friend bool operator!=(const xor_combine_engine &,
                         const xor_combine_engine &);

  // public data members
  static const bool has_fixed_range;
  static const int shift1;
  static const int shift2;
};
```

Description

Instantiations of `xor_combine_engine` model a pseudo-random number generator . To produce its output it invokes each of the base generators, shifts their results and xors them together.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`xor_combine_engine` public construct/copy/destruct

1.
   ```
   xor_combine_engine();
   ```

   Constructors a `xor_combine_engine` by default constructing both base generators.

2.
   ```
   xor_combine_engine(const base1_type & rng1, const base2_type & rng2);
   ```

   Constructs a `xor_combine` by copying two base generators.

3.
   ```
   explicit xor_combine_engine(result_type v);
   ```

   Constructs a `xor_combine_engine`, seeding both base generators with `v`.

   > **Warning**
   >
   > The exact algorithm used by this function may change in the future.

4.
   ```
   template<typename SeedSeq> explicit xor_combine_engine(SeedSeq & seq);
   ```

   Constructs a `xor_combine_engine`, seeding both base generators with values produced by `seq`.

5.
   ```
   template<typename It> xor_combine_engine(It & first, It last);
   ```

   Constructs a `xor_combine_engine`, seeding both base generators with values from the iterator range [first, last) and changes first to point to the element after the last one used. If there are not enough elements in the range to seed both generators, throws `std::invalid_argument`.

`xor_combine_engine` public member functions

1.
   ```
   void seed();
   ```

   Calls `seed()` for both base generators.

2.
   ```
   void seed(result_type v);
   ```

   `seeds` both base generators with `v`.

3.
   ```
   template<typename SeedSeq> void seed(SeedSeq & seq);
   ```

   `seeds` both base generators with values produced by `seq`.

4.
   ```
   template<typename It> void seed(It & first, It last);
   ```

   seeds both base generators with values from the iterator range [first, last) and changes first to point to the element after the last one used. If there are not enough elements in the range to seed both generators, throws `std::invalid_argument`.

5.
   ```
   const base1_type & base1() const;
   ```

   Returns the first base generator.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6.
```
const base2_type & base2() const;
```

Returns the second base generator.

7.
```
result_type operator()();
```

Returns the next value of the generator.

8.
```
template<typename Iter> void generate(Iter first, Iter last);
```

Fills a range with random values

9.
```
void discard(boost::uintmax_t z);
```

Advances the state of the generator by `z`.

`xor_combine_engine` public static functions

1.
```
static result_type min();
```

Returns the smallest value that the generator can produce.

2.
```
static result_type max();
```

Returns the largest value that the generator can produce.

`xor_combine_engine` friend functions

1.
```
template<typename CharT, typename Traits> 
  friend std::basic_ostream< CharT, Traits > & 
  operator<<(std::basic_ostream< CharT, Traits > & os, 
             const xor_combine_engine & s);
```

Writes the textual representation of the generator to a `std::ostream`.

2.
```
template<typename CharT, typename Traits> 
  friend std::basic_istream< CharT, Traits > & 
  operator>>(std::basic_istream< CharT, Traits > & is, 
             const xor_combine_engine & s);
```

Reads the textual representation of the generator from a `std::istream`.

3.
```
friend bool operator==(const xor_combine_engine & x, 
                       const xor_combine_engine & y);
```

Returns true if the two generators will produce identical sequences.

4.
```
friend bool operator!=(const xor_combine_engine & lhs, 
                       const xor_combine_engine & rhs);
```

Returns true if the two generators will produce different sequences.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Performance

For some people, performance of random number generation is an important consideration when choosing a random number generator or a particular distribution function. This page provides numerous performance tests with the wide variety of generators and distributions available in the boost library.

The performance has been evaluated on an Intel(R) Core(TM) i7 CPU Q 840 @ 1.87GHz, 1867 Mhz with Visual C++ 2010, Microsoft Windows 7 Professional and with gcc 4.4.5, Ubuntu Linux 2.6.35-25-generic. The speed is reported in million random numbers per second (M rn/sec), generated in a tight loop.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 12. Basic Generators (Linux)

generator	M rn/sec	time per random number [nsec]	relative speed compared to fastest [percent]
rand48	149.254	6.7	59%
lrand48 run-time	158.73	6.3	63%
minstd_rand0	22.9885	43.5	9%
minstd_rand	22.0751	45.3	8%
ecuyer combined	42.735	23.4	17%
kreutzer1986	151.515	6.6	60%
taus88	250	4	100%
knuth_b	19.6078	51	7%
hellekalek1995 (inversive)	4.54545	220	1%
mt11213b	204.082	4.9	81%
mt19937	204.082	4.9	81%
mt19937_64	60.6061	16.5	24%
lagged_fibonacci607	126.582	7.9	50%
lagged_fibonacci1279	129.87	7.7	51%
lagged_fibonacci2281	129.87	7.7	51%
lagged_fibonacci3217	131.579	7.6	52%
lagged_fibonacci4423	128.205	7.8	51%
lagged_fibonacci9689	128.205	7.8	51%
lagged_fibonacci19937	131.579	7.6	52%
lagged_fibonacci23209	131.579	7.6	52%
lagged_fibonacci44497	131.579	7.6	52%
subtract_with_carry	147.059	6.8	58%
subtract_with_carry_01	105.263	9.5	42%
ranlux3	15.748	63.5	6%
ranlux4	9.11577	109.7	3%
ranlux3_01	10.5708	94.6	4%
ranlux4_01	6.27353	159.4	2%

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

generator	**M rn/sec**	**time per random number [nsec]**	**relative speed compared to fastest [percent]**
ranlux64_3	15.8983	62.9	6%
ranlux64_4	9.14913	109.3	3%
ranlux64_3_01	10.9409	91.4	4%
ranlux64_4_01	6.32911	158	2%
ranlux24	15.1976	65.8	6%
ranlux48	8.88099	112.6	3%
mt19937ar.c	111.111	9	44%

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 13. Basic Generators (Windows)

generator	M rn/sec	time per random number [nsec]	relative speed compared to fastest [percent]
rand48	152.672	6.55	64%
lrand48 run-time	24.3724	41.03	10%
minstd_rand0	39.8248	25.11	16%
minstd_rand	39.0778	25.59	16%
ecuyer combined	16.7813	59.59	7%
kreutzer1986	89.0472	11.23	37%
taus88	237.53	4.21	100%
knuth_b	30.8166	32.45	12%
hellekalek1995 (inversive)	5.28457	189.23	2%
mt11213b	237.53	4.21	100%
mt19937	221.239	4.52	93%
mt19937_64	91.5751	10.92	38%
lagged_fibonacci607	142.45	7.02	59%
lagged_fibonacci1279	142.45	7.02	59%
lagged_fibonacci2281	145.56	6.87	61%
lagged_fibonacci3217	149.031	6.71	62%
lagged_fibonacci4423	142.45	7.02	59%
lagged_fibonacci9689	145.773	6.86	61%
lagged_fibonacci19937	142.45	7.02	59%
lagged_fibonacci23209	145.773	6.86	61%
lagged_fibonacci44497	142.45	7.02	59%
subtract_with_carry	136.24	7.34	57%
subtract_with_carry_01	90.3342	11.07	38%
ranlux3	13.1631	75.97	5%
ranlux4	7.60398	131.51	3%
ranlux3_01	8.62738	115.91	3%
ranlux4_01	4.99625	200.15	2%

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

generator	M rn/sec	time per random number [nsec]	relative speed compared to fastest [percent]
ranlux64_3	13.1631	75.97	5%
ranlux64_4	7.5861	131.82	3%
ranlux64_3_01	8.63931	115.75	3%
ranlux64_4_01	5.01958	199.22	2%
ranlux24	13.1631	75.97	5%
ranlux48	7.5861	131.82	3%
mt19937ar.c	200.401	4.99	84%

Note that the lagged Fibonacci and ranlux_01 generators produce floating-point numbers, whereas all others produce integers.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 14. Distributions (Linux)

[M rn/sec]	minstd_rand	kreutzer1986	mt19937	lagged_fibonacci607
uniform_int	16.2338	48.7805	21.5517	23.8663
uniform_smallint	18.9036	114.943	25.3165	74.6269
bernoulli	21.322	85.4701	23.2558	125
geometric	9.42507	11.7925	7.38007	15.528
binomial	13.4953	29.7619	12.7877	38.7597
negative_binomial	1.69549	2.29305	1.65563	2.45098
poisson	13.7552	34.1297	13.369	43.8596
uniform_real	18.2815	44.4444	19.8413	119.048
uniform_01	21.692	72.4638	17.1233	116.279
triangle	15.2207	29.3255	11.9904	51.2821
exponential	10.5374	17.0068	10.8814	22.2222
normal polar	8.82613	12.9199	9.00901	14.771
lognormal	6.15764	7.50188	5.68182	8.61326
chi squared	2.07297	2.8401	2.10926	3.07409
cauchy	9.18274	14.8368	7.37463	17.3913
fisher f	1.04646	1.47449	1.08026	1.61186
student t	1.60927	2.18245	1.65207	2.34192
gamma	2.1097	2.87439	2.13538	3.01296
weibull	4.73709	5.77367	4.20521	6.33312
extreme value	7.40192	10.101	6.23441	11.5741
uniform_on_sphere	2.22222	2.78552	2.28311	2.7933

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Table 15. Distributions (Windows)

[M rn/sec]	**minstd_rand**	**kreutzer1986**	**mt19937**	**lagged_fibonacci607**
uniform_int	27.049	79.1139	29.8151	34.8432
uniform_smallint	31.736	90.3342	33.9213	59.9161
bernoulli	25.641	56.2114	27.049	62.8141
geometric	12.8717	18.9645	14.6671	18.5805
binomial	18.2116	32.2165	19.8491	29.4118
negative_binomial	2.79065	3.99138	2.73358	3.72898
poisson	20.0321	37.7074	18.9645	36.4299
uniform_real	27.6319	78.1861	26.4901	71.2251
uniform_01	36.63	95.6938	26.3783	85.4701
triangle	19.4856	43.8982	19.425	36.8324
exponential	17.0474	32.0513	18.005	28.6205
normal polar	14.4051	19.7863	13.1354	20.7426
lognormal	10.8472	13.6968	10.3563	13.7855
chi squared	3.53957	4.95	3.44448	4.83442
cauchy	15.1906	23.5682	14.9768	23.31
fisher f	1.74951	2.45417	1.69854	2.38743
student t	2.63151	3.75291	2.53872	3.51432
gamma	3.50275	4.9729	3.35087	4.75195
weibull	8.96539	11.9161	9.09256	11.6754
extreme value	12.3274	18.4196	12.5945	17.5623
uniform_on_sphere	2.83688	3.58038	2.73898	3.60101

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

History and Acknowledgements

In November 1999, Jeet Sukumaran proposed a framework based on virtual functions, and later sketched a template-based approach. Ed Brey pointed out that Microsoft Visual C++ does not support in-class member initializations and suggested the enum workaround. Dave Abrahams highlighted quantization issues.

The first public release of this random number library materialized in March 2000 after extensive discussions on the boost mailing list. Many thanks to Beman Dawes for his original min_rand class, portability fixes, documentation suggestions, and general guidance. Harry Erwin sent a header file which provided additional insight into the requirements. Ed Brey and Beman Dawes wanted an iterator-like interface.

Beman Dawes managed the formal review, during which Matthias Troyer, Csaba Szepesvari, and Thomas Holenstein gave detailed comments. The reviewed version became an official part of boost on 17 June 2000.

Gary Powell contributed suggestions for code cleanliness. Dave Abrahams and Howard Hinnant suggested to move the basic generator templates from `namespace boost::detail` to `boost::random`.

Ed Brey asked to remove superfluous warnings and helped with `uint64_t` handling. Andreas Scherer tested with MSVC. Matthias Troyer contributed a `lagged Fibonacci generator`. Michael Stevens found a bug in the copy semantics of `normal_distribution` and suggested documentation improvements.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/