Random Number Generation in C++11

Document #: WG21 N3551
Date: 2013-03-12
Revises: None
Project: JTC1.22.32 Programming Language C++
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

For programmers seeking to familiarize themselves with the **<random>** component of the C++11 standard library, we provide background information and tutorial guidance with numerous usage examples.

1 Introduction

The production of random numbers has been an important application for computers since the beginning of the modern era of computation.[1] Alas, the C++ standard library has historically provided rather limited support, all of it adopted from the C standard library [Ker88, §7.8.7]. In header **<cstdlib>** (the C++ version of C's **<stdlib.h>**) we find:

- **RAND_MAX**, a macro that expands to an integer constant;
- **std::rand()**, a function that produces a pseudo-random number in the closed interval [0, **RAND_MAX**]; and
- **std::srand()**, a function to initialize (*seed*) a new sequence of such numbers.

[1] John von Neumann (né Neumann János Lajos, 1903–1957) is credited as the modern originator of the *middle-square method*, which he first described at a 1949 conference and published two years later [vNeu51]. Before that, von Neumann assisted Stanislaw M. Ulam (1909–1984) in developing the *Monte Carlo method*, still in heavy use today, for modelling complicated systems via random numbers and then performing statistical analyses of the resulting behavior.

The algorithms underlying even this minimal support have typically been unspecified, and hence their use has historically been nonportable, with oft-questionable performance and quality; "indeed, early C standard library implementations provided surprisingly bad generators" [Mau02].

To address the above shortcomings, the C++11 standard provides a new header, **<random>**. This article describes the technical underpinnings of this header and provides tutorial guidance and usage examples for programmers seeking to familiarize themselves with this significant component of the C++11 standard library.

2 Getting started

The traditional term *random number generator* encompasses (and unfortunately conflates) two kinds of functionality. However, C++11's **<random>** draws a clear distinction between the two:

- An *engine*'s role is to return unpredictable (*random*) bits,[2] ensuring that the likelihood of next obtaining a 0 bit is always the same as the likelihood of next obtaining a 1 bit.[3,4]
- A *distribution*'s role is to return random numbers (*variates*) whose likelihoods correspond to a specific shape. E.g., a *normal distribution* produces variates according to a "bell-shaped curve."

A user needs an object of each kind in order to obtain a random variate. The purpose of the engine is to serve as a *source of randomness*. We always use a distribution as well because "It makes no sense to ask for a random number without some context. You need to specify random according to what distribution" [Cook08]. Once we have an engine variable (let's name it **e**) and a distribution variable (let's name it **d**),[5] each call **d(e)** delivers one variate.[6]

The following function illustrates this simple three-step approach in simulating the roll of a fair (*unbiased*) die. First, line 5 defines an engine of type **default_random_engine**, as recommended by the standard for general use. To produce variates in a die's range, ensuring that each value has equal likelihood of appearing next, line 6 defines an appropriately-initialized distribution of type **uniform_int_distribution<int>**. Finally, line 7 obtains and returns the desired variate:

```
1 #include <random>
2
3 int  roll_a_fair_die( )
4 {
5   static std::default_random_engine         e{};
6   static std::uniform_int_distribution<int> d{1, 6};
7   return d(e);
8 }
```

[2] Unpredictability is the ideal. Using a computer, we generally settle for very-very-very-very-hard-to-predict (*pseudo-random*) bits.

[3] Using mathematical probability notation, this *uniformity* property can be compactly expressed as $p(0) = p(1)$.

[4] To reduce overhead and improve throughput, C++ engines typically compute and return many bits per call by encoding them into a single result of an unsigned integer type. For an engine of type **E**, these uniformly distributed unsigned values will lie in the closed range [**E::min()**, **E::max()**]. The number, b, of bits of randomness per call can be calculated as $b = \log_2(\texttt{E::max()} - \texttt{E::min()} + 1)$.

[5] The terms *engine* and *distribution* are applied not only to types meeting certain requirements specified in the C++11 standard, but also to objects of such types.

[6] Note that our code doesn't call engine **e** directly. Instead, the distribution **d** will do so, when and as often as needed, on our behalf.

3 An anti-pattern

Noting that an engine's results are already uniformly distributed, casual programmers often believe that it's not necessary to use any **`uniform_int_distribution`**. Instead, they transform their engine's output via their own algorithm in order to obtain variates in their desired range.

Here is a simple yet typical example[7] of such a misguided attempt to model the roll of a single fair die:

```
int  roll_a_biased_die( )
{
  static std::default_random_engine e{};
  return 1 + e() % 6;  // wrong!
}
```

Seemingly attractive, this function nonetheless is subtly wrong for two reasons:

- The function assumes that the engine's range spans at least six different values. Although most engines do span more than six values (and many span far more), an engine's range is technically required to span only two values.
- More importantly, "[the remainder] operation does not generate uniformly distributed random numbers (since in most cases this operation makes lower numbers slightly more likely)" [Cpp].

In either case, the function produces unfair (*biased*) results because its resulting values are incorrectly distributed.[8]

Some believe that the above anti-pattern is still "good enough" for casual use, but we respectfully disagree. At best, code such as shown above is sloppy; at worst, it gives erroneous results that are extremely hard to detect. Moreover, the code is likely over time to find its way into other, less tolerant, applications. We recommend avoiding such questionable code in the first place. Even in the rare cases that an engine's output actually happens to match a desired distribution, the absence of an explicit distribution forces each future reader of the code to expend mental energy verifying application correctness despite such an obvious lack.

Better still, standard-conforming distributions cope with all engine corner cases, even unlikely ones. For example, if an engine delivers fewer bits per call than needed, the distribution will make multiple calls until it has enough bits to let it satisfy the entire desired range of variates. On the other hand, if an engine produces more bits than needed, a distribution is free to cache the excess for use the next time it is called.

4 Initializing an engine

The process of initialization is often characterized to non-programmers as a "warming-up" stage. Properly initializing (*seeding*) an engine object can be critical to its subsequent correct behavior.

As we did above, an engine can be default-initialized. In this case, an engine's initial state is defined by its default constructor, whether invoked implicitly or explicitly:

```
std::default_random_engine  e1;   // implicitly default-initialized
std::default_random_engine  e2{}; // explicitly default-initialized
```

As one alternative, an engine's initial state can be influenced by providing an explicit starting value (a *seed*) to its constructor:

[7] To conserve space, we assume the **`#include <random>`** directive in this and most of our remaining code examples.

[8] See [Koe00, §7.4.4] for a somewhat expanded discussion and a function that copes with the bias via a simple *rejection* algorithm.

```
std::default_random_engine  e3{13607};
```

When value-initialized in this way, the type of the seed must be the same as (or convertible to) the type of the values produced by the engine. For an engine **e** of type **E**, this type is available as **decltype(e())**.[9]

When a program is run multiple times, its engine will always emit the same sequence of uniform bits if the engine is always initialized to the same state. Such *replication* can be very useful while debugging a program or reproducing research, for example. In other contexts, replication can be undesireable and even problematic.

To escape replication, the engine's seed must vary from run to run. Traditional approaches to this seeding subproblem involve the use of the system clock (e.g., via **time(0)**) or access to other system-specific sources of entropy (e.g., via **/dev/urandom**, where available). Programmers can avoid the need for such extra-linguistic facilities by using an object of type **random_device** to obtain a seed:[10]

```
std::random_device          rdev{};
std::default_random_engine e{rdev()};
```

Finally, note that an engine's state can be reset any time after it has been initialized. Calling a **seed()** member forces the engine into the same state it would have had upon corresponding initialization:

```
void  f( std::default_random_engine & e4, std::random_device & rd )
{
  e4.seed(13607);  // e4 state now as if initialized by 13607
  ...
  e4.seed();       // now as if default-initialized
  ...
  e4.seed(rd());   // now as if initialized via the device-obtained seed
}
```

5 What else can an engine do?

An engine has additional important capabilities not yet mentioned. For example:

- Engines are streamable, allowing save and restore, via operators **<<** and **>>**.
- Engines of the same type can be compared for equality/inequality.
- Engines can be seeded in still more ways than described above.
- Engines can skip (**discard**) generated values.

Because no distribution uses any of these extras, the standard library takes a two-tiered approach in specifying required behavior:

- A type that meets a distribution's modest needs is known as a URNG (Uniform Random Number Generator);
- A URNG that also has all the extra capabilities is then termed an engine.

[9] There are also two alternatives: **std::result_of<E()>::type**, which is a bit wordier, and **typename E::result_type**, the C++98-style interface that was the only option when **<random>** was first proposed in 2002. With the advent of **decltype** in C++11, the old-style **result_type** forms are being considered for deprecation throughout the library in a future revision of the C++ standard.

[10] It is worth keeping in mind that "there's no guarantee about *which* random device you get.... Implementations can define a string argument to the constructor that selects between the choices, but the default is supposed to be decent" [Yass11].

Except for **std::random_device**, all URNGs provided in the C++11 standard library are also engines.

The C++11 random number facility was also designed to be extensible. The standard carefully specifies the requirements so that knowledgeable users may devise and provide URNGs and engines of their own and have their types seamlessly interoperate with **<random>** and the rest of the standard library. Indeed, several new random number engines have been described in the recent literature, accompanied by sample C++ implementations that satisfy these requirements.[11]

6 Engines in the standard library

The C++11 standard library provides, in namespace **std**, engine types aimed at several different kinds of users.

- The library provides **default_random_engine**, an alias for an engine type selected by the library vendor, according to the standard, "on the basis of performance, size, quality, or any combination of such factors, . . . for relatively casual, inexpert, and/or lightweight use." Because distinct vendors are free to select differently, code that uses this alias need not generate identical sequences across implementations.
- For knowledgable users, the library provides nine additional aliases for pre-configured engine types of known good quality.[12] These types span a wide spectrum of trade-offs in performance and size. Among other characteristics, the C++11 standard requires that these engines produce bit-for-bit identical results across implementations.
  - Linear congruential engines: **minstd_rand0**, **minstd_rand**
  - Mersenne twister engines: **mt19937**, **mt19937_64**
  - Subtract with carry engines: **ranlux24_base**, **ranlux48_base**
  - Discard block engines: **ranlux24**, **ranlux48**
  - Shuffle order engine: **knuth_b**
- For experts (e.g., researchers), the library provides the following templates that can be configured via template parameters[13] (and in some cases then combined) to provide additional engine types.
  - **linear_congruential_engine**
  - **mersenne_twister_engine**
  - **subtract_with_carry_engine**
  - **discard_block_engine**
  - **independent_bits_engine**
  - **shuffle_order_engine**

7 Sharing an engine

It is common to find that an application needs to use an engine in more than one place. Rather than creating multiple independent engines, one to each purpose, a single engine can be created and shared (made available) wherever needed.[14] Perhaps the simplest way to do this is to make the engine local to a function that, when called, grants access to that engine:

---

[11] See, for example, [Sal11].

[12] The algorithmic characteristics of these engines (and also of the distributions described in the next section) have been carefully studied for many years. Their properties have become well understood, and have been described in numerous articles and books. For the reader interested in such details, we recommend [Knu97, chapter 3] as a starting point.

[13] Even experts should keep in mind that "Very few combinations of parameters actually result in engines of decent quality, but the committee philosophically leaned toward generality.... The point is that the concrete engines provided by the standard have known (and good) properties, while the [non-expert] user who fools with parameters other than these should know that he is doing something foolish" [Fis10].

[14] A hybrid approach is, of course, also possible. For example, each of several threads can create a thread-local engine and share it with the functions in that thread only.

```
std::default_random_engine &  my_engine( )
{
  static std::default_random_engine<>  e{};
  return e;
}
```

Such a function can be enhanced with appropriate synchronization if the engine is to be shared among multiple threads.

We can then use this function to emulate, for example, the C-style **rand**, **srand**, and **RAND_MAX** interface:[15]

```
#define RAND_MAX (my_engine().max() - my_engine().min())
void  srand( unsigned s = 1u )  { my_engine().seed(s); }
int  rand( )  { return my_engine()() - my_engine().min(); }
```

8 Distributions in the standard library

The C++11 standard library provides, in namespace **std**, twenty distribution types in five broad categories. While there are many, many other useful distributions, the following were selected for standardization based on the criteria presented in [Pat04].

- Uniform distributions:
  - **uniform_int_distribution**
  - **uniform_real_distribution**
- Bernoulli distributions:
  - **bernoulli_distribution**
  - **binomial_distribution**
  - **geometric_distribution**
  - **negative_binomial_distribution**
- Poisson distributions:
  - **poisson_distribution**
  - **exponential_distribution**
  - **extreme_value_distribution**
  - **gamma_distribution**
  - **weibull_distribution**
- Normal distributions:
  - **normal_distribution**
  - **cauchy_distribution**
  - **chi_squared_distribution**
  - **fisher_f_distribution**
  - **lognormal_distribution**
  - **student_t_distribution**
- Sampling distributions:
  - **discrete_distribution**
  - **piecewise_constant_distribution**
  - **piecewise_linear_distribution**

Most of these distributions are specified as class templates whose template parameter is the type of the variate to be produced. Some distributions are designed to deliver variates of only integer types, while others are designed to deliver variates of only floating-point types. The former is exemplified by **discrete_distribution**, while the latter is exemplified by **uniform_real_distribution**. The **bernoulli_distribution** is designed to deliver only **bool** variates, and is the only standard library distribution that is a class rather than a class template.

In much the same way as was done for URNG and engine types, the C++11 standard specifies the minimum requirements for distribution types. By adhering to these requirements, users can provide their own distributions and be confident that these distributions will seamlessly interoperate with any URNG, whether provided by the user or by the standard library. The net effect is that any URNG can be used with any distribution.

[15] We do not recommend using this interface; we present it here only as an illustration of the techniques described.

To select among the available distributions, it is typically necessary to begin with a firm understanding of the random process being modelled. For example, rolling a single die follows a uniform distribution as shown earlier, but rolling a pair of dice follows a different distribution:

```
int  roll_2_fair_dice( )
{
  static std::default_random_engine   e{};
  static std::discrete_distribution<> d{ { 1.0  // weight( 2)
                                         , 2.0  // weight( 3)
                                         , 3.0  // weight( 4)
                                         , 4.0  // weight( 5)
                                         , 5.0  // weight( 6)
                                         , 6.0  // weight( 7)
                                         , 5.0  // weight( 8)
                                         , 4.0  // weight( 9)
                                         , 3.0  // weight(10)
                                         , 2.0  // weight(11)
                                         , 1.0  // weight(12)
                                       } };

  return 2 + d(e);
}
```

It seems clear that we could produce a family of analogous functions if we need to simulate the roll of three dice, of four dice, etc. However, let's consider a different approach in which the previous two dice-rolling functions are consolidated so as to obtain a function simulating the roll of an arbitrary number of dice, specified as the function's argument.

The following example employs such a consolidated approach to demonstrate (a) that a single engine object can serve as a source of randomness for several distributions, and (b) that each distribution is initialized according to its own requirements.

```
int  roll_fair_dice( std::size_t n_dice )
{
  static std::default_random_engine      e{};
  static std::uniform_int_distribution<> d1{1, 6};
  static std::discrete_distribution<>    d2{ { 1.0  // weight( 2)
                                             , 2.0  // weight( 3)
                                             , 3.0  // weight( 4)
                                             , 4.0  // weight( 5)
                                             , 5.0  // weight( 6)
                                             , 6.0  // weight( 7)
                                             , 5.0  // weight( 8)
                                             , 4.0  // weight( 9)
                                             , 3.0  // weight(10)
                                             , 2.0  // weight(11)
                                             , 1.0  // weight(12)
                                           } };

  auto roll_1_fair_die  = [&] { return d1(e); };
  auto roll_2_fair_dice = [&] { return 2 + d2(e); };

  int total{ n_dice % 2u == 1u ? roll_1_fair_die()
                               : 0
           };
  for(  ;  n_dice > 1u;  n_dice -= 2u )
```

```
25    total += roll_2_fair_dice();
26   return total;
27 }
```

9 What else can a distribution do?

As illustrated above, a distribution is instantiated with parameters that are unique to the distribution's type. For example, a **uniform_int_distribution** has lower and upper bounds as parameters, while the parameters of a **normal_distribution** consist of the desired mean and standard deviation instead. Each call to a distribution uses the arguments with which the distribution was initialized to produce variates tailored to the application.

Each conforming distribution has an associated nested type, **param_type**, that serves to bundle a collection of initial values for that distribution. A **param_type** object always has the same parameters as its corresponding distribution, and can thus be constructed the same way:

```
1 using dist_t  = std::uniform_int_distribution<>;
2 using param_t = dist_t::param_type;
3
4 dist_t  d{1, 6};
5 param_t p{1, 6};
```

Just as a conforming engine can be reseeded, each conforming distribution allows its parameters' values to be adjusted permanently or temporarily. To make a temporary parameter adjustment, we construct a **param_type** object and then pass it as part of the call to the distribution:

```
1  int  roll_n_sided_die( int n )
2  {
3    using engine_t = std::default_random_engine;
4    using dist_t   = std::uniform_int_distribution<>;
5    using param_t  = dist_t::param_type;
6
7    static engine_t e{};
8    static dist_t   d{1, 6};
9
10   param_t p{1, n};
11   return d(e, p);
12 }
```

To make a permanent parameter adjustment, we construct a **param_type** object and pass it to the distribution's **param** function:

```
1 std::normal_distribution<> d{0.0, 1.0};
2 ...
3 using param_t = std::normal_distribution<>::param_type;
4 param_t p{0.0, 0.2};
5 d.param(p);  // use new parameters henceforth
```

Finally, each conforming distribution can be queried to obtain the values of its current parameters. Individual parameters can be obtained in a distribution-dependent manner; for example, a **normal_distribution** has members **mean()** and **stddev()** for such a purpose, while

a **uniform_int_distribution** has members **a()** and **b()**. Alternatively, a call to **param()** (with no arguments) will obtain a **param_type** bundle of all the current parameter values.

10 A simple toolkit

We present the following four functions as an example that not only combines many of the techiques described above, but that also provides a coherent toolkit of some of of **<random>**'s most commonly-used features. In the spirit of **default_random_engine**, these functions are likewise intended for "casual, inexpert, and/or lightweight use" and are recommended as replacements for **std::rand()** and its friends:

- **global_urng()**
  Shares a single URNG with the other functions in this toolkit.
- **randomize()**
  Sets the shared URNG to an unpredictable state.
- **pick_a_number(from,thru)**
  Returns an **int** variate uniformly distributed in [**from**, **thru**].
- **pick_a_number(from,upto)**
  Returns a **double** variate uniformly distributed in [**from**, **upto**).

```
1  std::default_random_engine &  global_urng( )
2  {
3    static std::default_random_engine  u{};
4    return u;
5  }

7  void  randomize( )
8  {
9    static std::random_device  rd{};
10   global_urng().seed( rd() );
11 }

13 int  pick_a_number( int from, int thru )
14 {
15   static std::uniform_int_distribution<>  d{};
16   using  parm_t  = decltype(d)::param_type;
17   return d( global_urng(), parm_t{from, thru} );
18 }

20 double  pick_a_number( double from, double upto )
21 {
22   static std::uniform_real_distribution<>  d{};
23   using  parm_t  = decltype(d)::param_type;
24   return d( global_urng(), parm_t{from, upto} );
25 }
```

11 A final example

While the following example exploits a number of features new in C++11, note especially the **std::shuffle()** algorithm. A variant of C++03's **std::random_shuffle()**, **std::shuffle()** is designed to use a C++11 URNG[16] as its source of randomness. The appropriate distribution is

[16] Recall from §5 that each conforming engine also satisifies the requirements of a URNG type.

internal to the algorithm, so no distribution need be provided by users. We also employ parts of the toolkit presented in the previous section:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <random>

int
  main( )
{
  // Manufacture a deck of cards:
  using card = int;
  std::array<card,52>  deck{};
  std::iota(deck.begin(), deck.end(), 0);

  // Shuffle the deck:
  randomize();
  std::shuffle(deck.begin(), deck.end(), global_urng());

  // Display each card in the shuffled deck:
  auto suit = []( card c ) { return "SHDC"[c / 13]; };
  auto rank = []( card c ) { return "AKQJT98765432"[c % 13]; };
  for( card c : deck )
    std::cout << '␣' << rank(c) << suit(c);
  std::cout << std::endl;
}
```

12 *Caveat lector!*

An early specification of **<random>** was published in [ISO07] in order to make it accessible to a wider community of users and to obtain their feedback toward the facility's evolution. In retrospect, the process worked as intended, and the now-standard version is as a result much improved over that early version in many important ways.[17]

Of course, in the interim a number of publications were produced by early adopters, commenting on and offering advice regarding the then-new facility. Because many of the later improvements to **<random>** rely on new C++11 core language features or are otherwise not backward-compatibile, the technical content of early writings (and, alas, of even some recent ones!) is often now at least somewhat suspect.

Moreover, in our experience, many **<random>**-related blogs and articles (a) give programming advice based on what seems to be a questionable understanding of the meaning, behavior, and properties of randomness, and (b) express opinion (commonly presented as fact) based on what seems to be a questionable understanding of **<random>**'s design. For all these reasons, we

[17] Many of the differences between the early and the final versions of **<random>** are described in [Bro06].

recommend that readers carefully consider the sources and dates of such publications before relying on them, especially in a C++11 context.

13 What's next?

With C++14 (and even C++17!) on the horizon, three distinct **<random>**-related adjustments are under consideration for future standardization. Here's a very brief overview; please see [Bro13] for the details of these proposals:

- Augmenting the **<algorithm>** header with **sample**, a well-known function to obtain a random sample of a requested size from a population whose size might be unknown.
- Augmenting the **<random>** header with a few novice-friendly functions much like those in the small toolkit shown above.
- Deprecating the use of **std::rand()** and its friends, a first step toward removing these functions (in a decade or so) from the **std** namespace.

14 Acknowledgments

Many thanks to the readers of early drafts of this paper for their helpful and insightful feedback.

15 Bibliography

[Bro06] Walter E. Brown, et al.: "Improvements to TR1's Facility for Random Number Generation." ISO/IEC JTC1/SC22/WG21 document N1933 (pre-Berlin mailing), 2006-02-23. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1933.pdf.

[Bro13] Walter E. Brown: "Three **<random>**-related Proposals." ISO/IEC JTC1/SC22/WG21 document N3547 (pre-Bristol mailing), 2013-03-08. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3547.pdf.

[Cook08] John D. Cook: "Pitfalls in Random Number Generation." 2008-10-23. http://www.codeproject.com/KB/recipes/pitfalls_random_number.aspx.

[Cpp] cplusplus.com: "function **rand**." Undated. http://www.cplusplus.com/reference/cstdlib/rand/.

[DuT12] Stefanus Du Toit: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N3485 (post-Portland mailing), 2012-11-02. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3485.pdf.

[Fis10] Mark S. Fischler: Private correspondence. 2010-05-27.

[ISO07] International Organization for Standardization: "Information technology — Programming languages — Technical Report on C++ Library Extensions." ISO/IEC document TR 19768:2007.

[Ker88] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language (Second Edition: ANSI C)*. Prentice-Hall, 1988. ISBN: 0-131-10370-9.

[Knu97] Donald E. Knuth: *Seminumerical Algorithms (Third Edition)*. Addison-Wesley, 1997. Volume 2 of *The Art of Computer Programming*. ISBN 0-201-89684-2.

[Koe00] Andrew Koenig and Barbara E. Moo: *Accelerated C++: Practical Programming by Example*. Addison-Wesley, 2000. ISBN 0-201-70353-X.

[Mau02] Jens Maurer: "A Proposal to Add an Extensible Random Number Facility to the Standard Library." ISO/IEC JTC1/SC22/WG21 document N1398 (post-SantaCruz mailing), 2002-11-10. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1398.html.

[Pat04] Marc Paterno: "On Random-Number Distributions for C++0x." ISO/IEC JTC1/SC22/WG21 document N1588 (pre-Sydney mailing), 2004-02-13.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1588.pdf.

[Sal11] John K. Salmon, et al.: "Parallel Random Numbers: As Easy as 1, 2, 3." In *SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, 2011. ISBN: 978-1-4503-0771-0.

[vNeu51] John von Neumann: "Various Techniques Used in Connection with Random Digits." In Householder, A. S. et al., eds. *Monte Carlo Method, National Bureau of Standards Applied Mathematics Series*, vol. 12. U. S. Government Printing Office, 1951.

[Yass11] Jeffrey Yasskin: Untitled response to posted question. In "generating random numbers in C++ using TR1 /dev/random . . . ," 2011-12-28.
http://stackoverflow.com/questions/8661231/generating-random-numbers-in-c-using-tr1-dev-random-resilient-to-1-second-r.

16 Revision history

Revision	Date	Changes
1.0	2013-03-12	• Published as N3551.