Random Number Generation is Not Simple!

Document #: WG21 N3847
Date: 2014-01-01
Revises: None
Project: JTC1.22.32 Programming Language C++
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

We discuss several approaches to the modern generation of random numbers using the facilities in the C++11 standard library header **<random>**, and solicit feedback from LEWG and LWG as to which, if any, are suitable candidates for future standardization efforts.

O!, many a shaft at random sent / Finds mark the archer little meant!
And many a word at random spoken / May soothe, or wound, a heart that's broken!
— SIR WALTER SCOTT

1 Introduction

There seems to be a pervasive belief among journeyman and even expert C++ programmers that (1) C++98 and C++03 provided a random number generator[1] and that (2) C++11 and C++14 additionally provide a selection of random number generator types.[2] Alas, this belief is untrue: No entity in the standard library is labelled a *random number generator*, and no single library entity can take the place of one except under some very specific circumstances.

There seems to be a further persistent belief, among inexperienced programmers and *cognoscenti* alike, that random number generation is (or at least ought to be) "simple." However, our experience as well as our more than decade-long study of the random-number-generation literature of the past 75 years (e.g., [JvN51]) suggests quite the contrary, hence this paper's title. Historically, it has been largely a question of the most suitable algorithms for obtaining (and validating) pseudo-randomness properties, and the standard library's **<random>** facilities were of course designed to help its users with those aspects. But there's far more to the tale!

[1] Namely, function **rand()** in header **<cstdlib>**.

[2] Namely, the engine templates in header **<random>**. Recall that the requirements of a *random number engine* type are a strict superset of the requirements of a *uniform random number generator* type; see [rand.req.eng]/1 ff.

This paper will discuss several approaches to the modern generation of random numbers using the facilities in the C++11 standard library header **<random>**. Before doing so, we conjecture how suboptimal nomenclature may have influenced such incorrect conclusions on the part of so many.

2 Nomenclature matters

At the time we were designing what became the C++11 random number facility, we had become somewhat concerned about the amount of nomenclature we were introducing. As part of an effort to keep at least some generally familiar terminology, we were induced to recycle the conventional term "random number generator." We combined it with a descriptive adjective to obtain *uniform random number generator* (URNG for short), a term of art now used in clause 26 to denote a certain set of requirements. In particular, those requirements specify the algorithmic interface for types and objects that produce sequences of bits in which each possible bit value is uniformly likely.[3]

A single call to a URNG object is allowed to produce and deliver many (typically 32 or more) bits, returning these bits as a single packaged value of an unsigned integer type.[4] Experimentation showed that this design consistently produces a considerable performance improvement when compared to our earliest (unpublished) efforts in which each call to a URNG object yielded but a single bit. However, the adopted bit-production-in-bulk design now appears to be misleading many programmers as to a URNG's intended use.

What we have observed over the last few years is that many programmers mistakenly use a URNG[5] as if it were a *random number distribution*, a term of art that C++ uses (since TR1 [ISO07]) to specify a very different set of requirements.[6] When invoked, a distribution object produces what is known in probability and statistics as a *random variate*; in the computing disciplines, such a variate has long been known as a *random number*. We conjecture that it is the extensive tradition behind the *random number* term that is misleading many programmers as to the purpose and correct use of what we termed a URNG.

With benefit of hindsight, we wish we had used a slightly different term in place of URNG, a term that more accurately described the designed purpose of such types and objects as sources of randomness, not as sources of random variates. Accordingly, we will hereinafter use the similar but more precise term *uniform random bit generator* (abbreviated URBG) in place of *uniform random number generator*.[7] With this small adjustment in nomenclature, it should be much clearer that there is no entity in the standard library that by itself directly corresponds to the traditional general notion of a random number generator.

3 Basic approaches

3.1 The intended pattern

For C++11 and C++14, the functionality typically expected of a *random number generator* is most simply obtained by:

---

[3] That is, each generated bit is exactly as likely to be a 0 as it is to be a 1.

[4] Conceptually, we wanted many of the semantics a **vector<bool>** would give us. Since many of the algorithms we were standardizing already used similar encodings, we opted to mimic part of a typical **vector<bool>** implementation.

[5] Calls to **rand()** have the same issues; see [Walker].

[6] Please see our earlier paper [N3551] for additional exposition regarding the roles of URNGs and distributions, their interaction, and the all-too-common anti-pattern that misuses a URNG.

[7] With LWG endorsement, we would be willing to submit an editorial proposal to adjust nomenclature in library clauses 25 and (mostly) 26 along the following lines: *uniform random ~~number~~ bit generator*, **UniformRandom~~Number~~BitGenerator**, and UR~~N~~BG.

1. instantiating an object `u` of a type `U` that satisfies the requirements of a URBG;[8]
2. instantiating an object `d` of a type `D` that satisfies the requirements of a *random number distribution*; and
3. thereafter calling `d(u)`, as often as desired, to obtain random variates.

The following function from [N3551] illustrates one form of the recommended pattern:[9]

```
int  roll_a_fair_die( )
{
  static default_random_engine         e{};
  static uniform_int_distribution<int> d{1, 6};
  return d(e);
}
```

Note that such a function is also well-suited for use with standard library algorithms:

```
constexpr size_t N = 1000;
int a[N];
generate_n(a, N, roll_a_fair_die);
```

3.2 Common anti-patterns

In our experience, user code that makes any direct call `u()` to a URBG object is nearly always buggy. In order to be okay, it is almost always the case that the application must use (or discard) the generated value as-is, i.e., without any transformation. The following variant of the above function's body exemplifies a typical naïve[10] attempt to obtain uniformly-distributed numbers in a different range via a simple transformation that, alas, "produces unfair (*biased*) results because its resulting values are incorrectly distributed" [N3551]:

```
  static default_random_engine e{};  // same as above
  return 1 + e() % 6;  // wrong--no longer uniformly distributed!
```

Discussing very similar code, Julienne Walker points out:

> Anyone who [applies this remaindering technique] will be rewarded with a seemingly random sequence and be thrilled that their clever solution worked. Unfo[r]tunately, this does not work [. . . .] because forcing the range in this way eliminates any chance of having a uniform distribution. Now, this is okay if you care nothing about some numbers being more probable than others, but to be correct, you must work with the distribution instead of destroy it [Walker].

3.3 The lesson

If an application requires uniformly-distributed random variates in a URBG's exact range of bulk values, namely from `U::min()` through `U::max()`, then working with a URBG as if it were a distribution *might* be okay. However, "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" [N3551]. Most importantly, though, is that any other requirement (e.g., for a

[8] Recall again that a *random number engine* type satisfies all requirements of a URBG type and more. We sometimes name our URBG objects `e` (rather than `u`) because most of the standard library's URBG types are also random number engine types.

[9] For simplicity, we omit `std::` qualifications as well as necessary headers throughout all code samples herein.

[10] "A naïve algorithm is a very simple solution to a problem. It is meant to describe a suboptimal algorithm compared to a 'clever' (but less simple) algorithm" [Atw07].

different range, a different type, a non-uniform distribution, etc.) should promptly be met by instantiating and calling an appropriately-initialized distribution object as recommended above.

4 A function toolkit

In [N3547] and its successor [N3742], we proposed "to add to **<random>** the following modest toolkit of novice-friendly functions:

- **global_urng()**
  "Grants access to a URNG object of implementation-specified type.
- **randomize()**
  "Sets the above URNG object to an unpredictable state.[11]
- **pick_a_number(from,thru)**
  "Returns an **int** variate uniformly distributed in the closed **int** range [**from**, **thru**].
- **pick_a_number(from,upto)**
  "Returns a **double** variate uniformly distributed in the half-open **double** range [**from**, **upto**)."

We had provided, in that proposal, the following sample implementation (here slightly edited for consistency with our revised URBG nomenclature as described above):

```
auto&  global_urbg( )
{
  static default_random_engine  u{};
  return u;
}
```

```
void  randomize( )
{
  static random_device  rd{};
  global_urbg().seed( rd() );
}
```

```
int  pick_a_number( int from, int thru )
{
  static uniform_int_distribution<>  d{};
  using parm_t  = decltype(d)::param_type;
  return d( global_urbg(), parm_t{from, thru} );
}

double  pick_a_number( double from, double upto )
{
  static uniform_real_distribution<>  d{};
  using parm_t  = decltype(d)::param_type;
  return d( global_urbg(), parm_t{from, upto} );
}
```

Such a toolkit follows the basic recommendation, but does so in a fashion that provides several advantages over the single-function approach illustrated earlier. For example, the URBG can at any time be reseeded (reinitialized):

- to a known state via **global_urbg().seed(···)** in order to ensure reproducibility, or
- to an unknown state via **randomize()** in order to avoid reproducibility.

[11] "Unpredictability is the ideal. Using a computer, we typically settle for very-very-very-very-hard-to-predict."

Further, a distribution is not limited to a single range of values, as the desired range is provided per call via **pick_a_number**'s function arguments. However, this design means the function is poorly suited for use with most standard library algorithms, as they tend to expect niladic function objects as arguments.

5 A function template toolkit

The overloaded **pick_a_number** functions shown in the previous section can be reformulated as function templates. This approach allows callers the additional freedom to specify their desired return type:

```
template< class T >
enable_if_t<is_integral<T>{}(), T>
  pick_a_number( T from, T thru )
{
  static uniform_int_distribution<T>  d{};
  using parm_t  = decltype(d)::param_type;
  return d( global_urbg(), parm_t{from, thru} );
}

template< class T >
enable_if_t<is_floating_point<T>{}(), T>
  pick_a_number( T from, T upto )
{
  static uniform_real_distribution<T>  d{};
  using parm_t  = decltype(d)::param_type;
  return d( global_urbg(), parm_t{from, upto} );
}
```

6 A class toolkit

The following approach features a class interface to random variate generation. Despite its simplicity of use for simple tasks, it offers considerable flexibility for configuring its internal URBG and distribution resources.

```
class random_number_source
{
public:
  // types
  using urbg_type         = default_random_engine;
  using distribution_type = uniform_int_distribution<int>;
  using seed_type   = typename urbg_type::result_type;
  using param_type  = typename distribution_type::param_type;
  using result_type = typename distribution_type::result_type;

private:
  urbg_type          e;
  distribution_type  d;

public:
  // construct
  random_number_source( ) = default;
  random_number_source( seed_type seed ) : e{seed}, d{}  { }
```

```
  // use compiler-generated copy/move/destroy

  // reinitialize
  random_number_source&
    seed( seed_type seed )  { e.seed(seed); d.reset(); return *this; }
  random_number_source&
    randomize( )  { return seed( random_device{}() ); }
  template< class P0, class... P1toN >
  param_type
    param( P0&& p0, P1toN&&... p1ton )
  {
    param_type  p = d.param();
    d.param( param_type( forward<P0>(p0)
                       , forward<P1toN...>(p1ton...)
                       )
           );
    d.reset();
    return p;
  }

  // produce random variate
  result_type
    operator () ( )  { return d(e); }
  template< class P0, class... P1toN >
  result_type
    operator () ( P0 p0, P1toN... p1ton )
  {
    return d(e, param_type( forward<P0>(p0)
                          , forward<P1toN...>(p1ton...)
                          )
            );
  }

  // observe
  urbg_type&          urbg( )  { return e; }
  distribution_type&  distribution( )  { return d; }

  // equality-compare
  bool
    operator == ( random_number_source const& other )
  { return e == other.e and d == other.d; }
  bool
    operator != ( random_number_source const& other )
  { return not (*this == other); }
};
```

Although an object of such a type is a niladic function object, it is still somewhat poorly suited for use with most standard library algorithms in all but the very simplest of use cases. The issue is that such stateful objects should nearly always be passed to an algorithm by reference, but of course this is not the default parameter-passage mechanism used by the standard library for function objects. It is therefore up to the user to wrap such an object **`g`** (for example, as **`ref(g)`**)

at each point of call, lest the URBG and distribution members be copied and so make possible unexpected duplicate sequences of variates.[12]

7 A class template toolkit

We note in passing that the class presented in the previous section can be reformulated as a class template. Such a variation provides users the additional capability of specifying their desired URBG and distribution types, yet providing as defaults those most commonly wanted. We show the template's initial part only, as the bulk of the code duplicates that of the class exhibited above:

```
template< class URBG         = default_random_engine
        , class Distribution = uniform_int_distribution<int>
        >
  class random_number_source
{
private:
  URBG          e;
  Distribution  d;

public:
  // types
  using urbg_type         = URBG;
  using distribution_type = Distribution;

// etc.
```

While this reformulation does provide some additional flexibility, users must still exercise care whenever passing an object of such a type, for the same reasons stated above.

8 An iterator toolkit

We now present yet another approach, featuring

- an iterator interface to the generation of random variates, and
- internal reference semantics to preempt issues caused by copying.

An iterator interface can be advantageous in connection with such algorithms as **copy_n**; here are several examples of such use:

```
using rdev_t    = random_device;
using urbt_t    = default_random_engine;
using dist_t    = uniform_real_distribution<double>;
using variate_t = dist_t::result_type;

urbt_t u{ rdev_t{}() };
dist_t d{ };
variate_iterator<urbt_t,dist_t> it{u, d};  // see below

// create a random 5 dimensional vector:
vector<variate_t>  v( 5 );
copy_n(it, 5, v.begin());
```

[12] See also the std-discussion newsgroup thread starting with "How to avoid accidental copying of random number generators" (Christopher Jefferson, 2013-12-03).

```
// dot product with a random vector:
variate_t prod{ inner_product( v.begin(), v.end(), it, 0.0) };

// add noise to a vector:
transform( v.begin(), v.end()
         , v.begin()
         , [&] ( variate_t d ) { return d + *it++; }
         );
```

Note that the use of reference semantics has again made the user responsible for instantiating and maintaining ownership of the underlying URBG and distribution. However, it is typical for iterators not to own their referents, so such responsibility should be no surprise to users:

```
template< class Distribution = uniform_int_distribution<int>
        , class URBG         = default_random_engine
        >
  class variate_iterator
: public iterator< input_iterator_tag, typename D::result_type >
{
private:
  using iter_t = variate_iterator;
  using val_t  = typename D::result_type;
  using ptr_t  = val_t const*;
  using ref_t  = val_t const&;

  URBG         *  u{ nullptr };  // non-owning
  Distribution *  d{ nullptr };  // non-owning
  val_t           v{ };
  bool            valid{ false };

  // help:
  void   step(  ) noexcept  { valid = false; }
  ref_t  deref( )           { if( not valid )
                                v = (*d)(*u), valid = true;
                              return v;
                            }
  bool   eq( iter_t const& o ) const noexcept
  { return ( (u == o.u and d == o.d)
          or (   u and o.u and *u == *o.u
             and d and o.d and *d == *o.d
           ) )
       and valid ? (v == o.v) : true
       and valid == o.valid;
  }

public:
  // construct:
  constexpr variate_iterator( ) noexcept = default;
  variate_iterator( URBG& u, Distribution& d ) : u{ &u }, d{ &d }  { }

  // use compiler-generated copy/move/destroy

  // dereference:
  val_t  operator *  ( )        { return  deref(); }
  ptr_t  operator -> ( )        { return &deref(); }
```

```
44    // advance:
45    iter_t&  operator ++ (     )  { step(); return *this; }
46    iter_t   operator ++ ( int )  { iter_t tmp{*this}; step(); return tmp; }

48    // equality-compare:
49    bool  operator == ( iter_t const& other ) const  { return     eq(other); }
50    bool  operator != ( iter_t const& other ) const  { return not eq(other); }
51  };
```

9 Summary and conclusion

We have shown several ways to take advantage of the facilities in **<random>**, each consistent with the basic approach recommended in §3. Factoring out the additional flexibility provided by template variants, we exhibited random variate generation via:

- free functions that instantiate and statically own URBG and distribution objects;
- a class type that instantiates and owns URBG and distribution member objects; and
- an iterator type that applies reference semantics in accessing user-owned URBG and distribution objects.

In each case, our generators apply a resource-management policy to the URBG and distribution objects needed to produce random variates. Although our sample code necessarily combines a choice of generator interface with a choice of management policy, it should be clear that any resource policy can be combined with any choice of interface, and that each combination could be advantageous to certain applications while suboptimal with respect to others.

Moreover, there are still other approaches that can be explored. For example, one could have a generic **generator_iterator** facility, capable of accepting any kind of generator object and providing an iterator interface to it. We have also not studied issues related to concurrency and parallelism, such as would arise in the generation of massive quantities of random variates. Striking an appropriate engineering balance in designing such tools seems not simple, indeed.

In addition, we have not explored at all some terribly important subproblems such as algorithms for selecting good URBG seeds. (Using the system **time()** has long been known as a poor seeding choice for any kind of serious work.) As a starting point for such a discussion, see the now-dated [Gar96, §23.8–9].

It is thus an open issue for WG21 to decide which, if any, approach is both suitably important and suitably general for possible future standardization. We note for reference that the status quo is not accidental, but is rather the result of a previous decision made by LWG and endorsed by WG21. While TR1 [ISO07] had specified a **variate_generator** utility template, this template was very deliberately not proposed for C++11: "with no loss of generality and with no loss of functionality, we decided to omit **variate_generator** from the [accompanying] Proposal" [N1933, §6: "The demise of **variate_generator**"].

We invite LEWG and/or LWG to review this past decision, along with the possible directions described in the present paper. As evidenced by WG21's recent and past anxiety and frustration about this and related issues, we recognize that this is not an easy decision. However, continued

vacillation is in no one's interests, so we urge a final determination either to pursue some specified direction for future standardization, or to pursue none.

10 Acknowledgments

Many thanks to the readers of early drafts of this paper for their thoughtful comments. Special thanks to Mike Spertus for suggesting, several years ago, the use cases presented in §8.

11 Bibliography

[Atw07] Jeff Atwood: "The Danger of Naïveté." In blog *Coding Horror*, 2007-12-07. http://www.codinghorror.com/blog/2007/12/the-danger-of-naivete.html.

[Gar96] Simson Garfinkel, Gene Spafford, and Simson Garfinkel: *Practical UNIX and Internet Security*, 2nd Edition. O'Reilly & Associates, 1996. ISBN 1565921488.

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

[JvN51] John von Neumann: "Various techniques used in connection with random digits." In *Proceedings of Symposium on "Monte Carlo Method"* (held June–July 1949 in Los Angeles). National Bureau of Standards, Applied Math Series 12, 1951-06-11, pp. 36–38.

[N1933] Walter E. Brown, Mark S. Fischler, Jim Kowalkowski, and Marc Paterno: "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.

[N3547] Walter E. Brown: "Three **<random>**-related Proposals." ISO/IEC JTC1/SC22/WG21 document N3547 (pre-Bristol mailing), 2013-03-12. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3547.pdf.

[N3551] Walter E. Brown: "Random Number Generation in C++11." ISO/IEC JTC1/SC22/WG21 document N3551 (pre-Bristol mailing), 2013-03-12. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3551.pdf.

[N3742] Walter E. Brown: "Three **<random>**-related Proposals, v2." ISO/IEC JTC1/SC22/WG21 document N3742 (pre-Chicago mailing), 2013-08-30. A revision of [N3547]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3742.pdf.

[Walker] Julienne Walker: "Using **rand()**." In blog *Eternally Confuzzled*, undated. http://eternallyconfuzzled.com/arts/jsw_art_rand.aspx.

12 Revision history

Version	Date	Changes
1	2014-01-01	• Published as N3847.