What C++ Programmers Need to Know about Header <random>

Expanded edition, with minor corrections applied

WALTER E. BROWN, PH.D.

< webrown.cpp @ gmail.com >

Overview

- During this talk, we will:
  - Present the conceptual and technical underpinnings of the <random> header,
  - ✓ Demonstrate its most common correct usage pattern,
  - X Show the most common usage anti-pattern and explain what's wrong with it,
  - Explore some <random>-based toolkit designs,
  - Have a little <random> fun, and more!
- Some of this material is adapted from my WG21 papers:
  - N3551: “Random Number Generation in C++11” (2013).
  - N3847: “Random Number Generation is Not Simple!” (2014).
- Slides hidden (for lack of time) will be available online.

A little about me

- B.A. (math's); M.S., Ph.D. (computer science).
- Professional programmer for ~~almost~~ 50 years, programming in C++ since 1982.
- Experienced in industry, academia, consulting, and research:
  - Founded a Computer Science Dept.; served as Professor and Dept. Head; taught and mentored at all levels.
  - Managed and mentored the programming staff for a reseller.
  - Lectured internationally as a software consultant and commercial trainer.
  - Retired from the Scientific Computing Division at Fermilab, specializing in C++ programming and in-house consulting.
- Not dead — still doing training & consulting. (Email me!)

Emeritus participant in C++ standardization

- Written 100+ papers for WG21, proposing such now-standard C++ library features as gcd/lcm, cbegin/cend, and common_type, as well as the entirety of headers <random> and <ratio>.
- Influenced such core language features as *alias templates*, *contextual conversions*, and *variable templates*; working on *requires-expressions*, operator synthesis, and more!
- Conceived and served as Project Editor for ISO/IEC 29124 (Int'l Standard on Mathematical Special Functions in C++), now incorporated into C++17's <cmath>.
- Be forewarned: Based on my training and experience, I hold some rather strong opinions about computer software and programming methodology — these opinions are not shared by all programmers, but they should be! ☺

Background Information re Randomness

Random numbers are a pain in the butt. However, they're terribly interesting, and very useful in many applications....

— Julienne Walker

Random numbers should not be generated with a method chosen at random.

— Donald Knuth

In the beginning

- The first known algorithm is the *middle-square method*:
  - Maybe discovered by Brother Edvin, a Franciscan, circa 1245.
  - Discovered by John von Neumann (né Neumann János Lajos, 1903–1957) circa 1949.
  - Considered crude but fast, with obvious failure modes.
- The *Monte Carlo method* (a code name) of simulation:
  - Developed 1946+ by Stanislaw M. Ulam (1909–1984), coded on ENIAC by von Neumann, for the Manhattan Project.
  - Still heavily used today for modelling complicated processes via random numbers …
  - Then statistically analyzing the resulting behavior.

Uses of randomness today

- Monte Carlo and other simulations
- Genetic algorithms, randomized algorithms, Zobrist and other hashing algorithms
- Rearranging (permuting, shuffling): • testing sort algorithms • “randomizing” tournaments (sporting events/military draft/ arranging candidates on voter ballots)
- Selection: • gaming (raffles/lotteries/slot machines) • calling on students fairly • aesthetics (art/music/poetry) • choosing performers on open-mike night • choosing quicksort’s pivot
- Statistical sampling: • mfg. quality control • drug screening • auditing • trials for new medical treatments/drugs • surveys
- Cryptography (has extra requirements not met by <random>)

☺

"You have reached the Heisenberg Institute. Please hold and your call will be answered in random order."

Just recognizing randomness can be difficult

- Imagine an oracle gave you ten zeroes in a row. Random?
- "[T]here's nothing suspicious about … a sequence of ten zeros. Ten values just isn't enough to draw any conclusions about the quality of a random number generator."

  — Pete Becker

- Expressed another way:

— Scott Adams

Intuition fails most of us re randomness

- "… recent research [by P. Diaconis *et al.*] into coin flips has discovered that … the chance of a coin landing in the same position as it started is about 51 percent. Heads facing up predicts heads; tails predicts tails."

  — David E. Adler

- "I watched a guy at a Vegas casino flip a coin to tell if he was to 'draw' another card at a blackjack table…. The coin landed on its edge….. You could hear a pin drop!"

  — Joe Peach

Achieving random behavior is astonishingly difficult

- "Statistical definitions [of randomness] involve the inability to predict outcomes or to find any pattern in a series of outcomes."

  — Chris Wetzel

- "Randomness is really, really hard for computers."

  — Jeff Atwood

- "[I]t's hard (and interesting) to get a computer to generate proper random numbers."

  — Mads Haahr

- The randomness literature (including textbooks!) is littered with algorithms that ultimately proved to be inadequate at best, or outright wrong at worst.

Training is sorely lacking

- Programming curricula rarely teach randomness.
- Random variates, widely thought to be simple, just aren't!
  - Example: if f( ) is uniformly distributed, so is 2 ∗ f( ); yet f( ) + f( ) is normally distributed, not uniformly.
  - So we must be careful with even simple refactorings!
- Programming puzzle (von Neumann, 1951):
  - Given a biased coin [unknown prob. $0 < p < 1$ of heads] …
  - How can we nonetheless algorithmically obtain fair results?
    - Solution in Wikipedia: "Fair coin" …
    - But first think about it, okay? ☺

Advice: tread extremely carefully!

- “Many generators have been written, most of them have demonstrably non-random characteristics, and some are embarrassingly bad.”

  — Stephen K. Park & Keith W. Miller

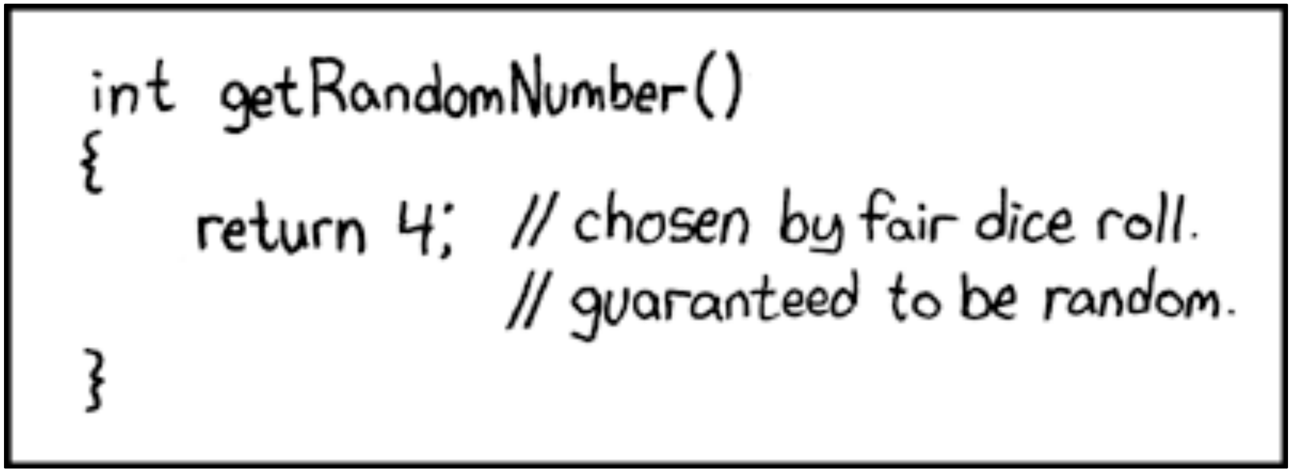
- “In the past, there have been some truly awful RNG algorithms published in well-known places, and then used by many people.”

  — Brad Lucier

- “The message … is: *arbitrary operations on random numbers do not necessarily result in random output.*”

  — detly (blog)

And then we find …



— Randall Munroe

(A classic case of
“exactly what you asked for,
but not what you wanted.”)

What about rand( ) (in <stdlib.h>/<cstdlib>)?

- C11 says (in a footnote):
  - "There are no guarantees as to the quality of the random sequence produced and …
  - "some implementations are known to produce sequences with distressingly non-random low-order bits. …"
- C++17 (Committee Draft, N4604) says (in a Note) that:
  - "rand's underlying algorithm is unspecified.
  - "Use of rand therefore continues to be nonportable, with unpredictable and oft-questionable quality and performance."
- "[<random>] replaces the C random library the same way a computer replaces an abacus." — Indi (blog)

What’s in Header <random>?

(Quite a Lot!)

Thanks to its uncompromising attention to generality and performance, one expert deemed [<random>] ‘what every random number library wants to be when it grows up.’

— Bjarne Stroustrup

<random> has something for everyone

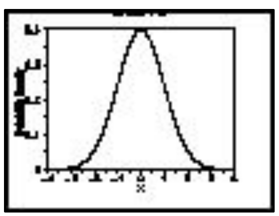
- For casual users:
  - 1 preconfigured engine type
  - 2 configurable uniform distribution class templates
- For intermediate & advanced users:
  - 9 more preconfigured engine types
  - 18 more configurable distribution class templates
  - 1 URBG class for environmental sources of randomness
  - 1 class to help with engine seeding
- For experts:
  - 6 configurable engine/engine adapter class templates
  - 1 function template for authors of new distributions

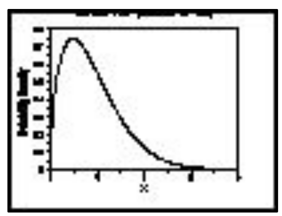
C++ nomenclature

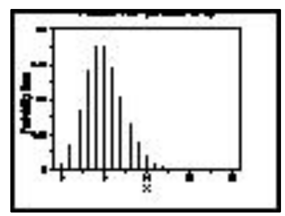
- Traditional term *random number generator* is unfortunate:
  - It blurs the distinction between two kinds of functionality …
  - So <random> uses more precise terminology.

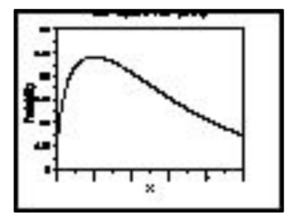
① An engine is a means of obtaining a sequence of (ideally) unpredictable bits that are uniform (*i.e.*, p(0) = p(1)).

② A distribution is a means of obtaining variates (values of desired aggregated shape) from such a sequence of bits.


Normal


Weibull


Poisson


Chi-square

Engine types and objects

- An engine object e of type E gives $n > 0$ bits per call e( ):
  - Once initialized, e( )'s output is deterministic, so is termed pseudo-random.
  - "Unpredictability is the ideal. Using a computer, we typically settle for very-very-very-very-hard-to-predict."
- Some details:
  - Each call e( ) produces a *bit string* of nonzero length *n* …
  - Encoded as a value of some unsigned E::result_type …
  - In the range [E::min( ) .. E::max( )], often [0 .. 2^n-1].
- But these details are rarely needed:
  - ☛ An engine object should be used nearly exclusively as a source of randomness (*e.g.*, for a distribution object).

Distribution types and objects

- A distribution object produces a variate consistent with the distribution's result type and c'tor arguments:
  - *E.g.*, std::uniform_int_distribution< long > d1{ low, high } ;
  - *E.g.*, std::normal_distribution< float > d2{ mean, stddev } ;
- Examples:
  - ```
    #include <random>
    int  roll_a_fair_die(  )  {
       static  std::default_random_engine          e{ } ;      // reproducible
       static  std::uniform_int_distribution<int> d{ 1, 6 } ;
       return  d(e);     // obtains a random variate
    }
    ```
  - ```
    constexpr  std::size_t  N = 1000;
    int  a[N];
    std::generate( a + 0, a + N, roll_a_fair_die );
    ```

<random> provides 10 pre-configured engine types

- Implementation-defined default_random_engine type:
  - Intended for "casual, inexpert, and/or lightweight use."
  - Vendor's alg. choice (based on performance, size, quality, …).
  - Needn't give the same sequence across platforms.
- The other 9 alg's are bit-for-bit portable across platforms:
  - 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

Why those 9 algorithms?

- Their characteristics (performance, size, quality, *etc.*) have been carefully studied for ~~many years~~ decades and are well described in standard references (*e.g.*, Knuth: *TAOCP* vol. 2):
  - *E.g.*, linear cong. is small and fast, but is of poorer quality …
  - Than Mersenne twister, whose state is >600x larger.
- Underlying engine templates are for researchers, *etc.*:
  - Configurable via template parameters to provide additional engine types, *e.g.*, linear_congruential_engine< ... >.
  - But “Very few combinations of [template] parameters actually result in engines of decent quality.... The [non-expert] user who fools with [these] parameters … should know that he is doing something foolish.” — Mark Fischler

Engines can do more

- Some additional engine capabilities:
  - Engines are streamable (*e.g.*, to allow reproducibility or resuming an interrupted computation).
  - Engines of the same type can be compared for (in)equality.
  - Engines can be seeded (and also reseed( )ed) in more ways.
  - Engines can discard( ) (skip) generated values.
- But distributions use none of those extras:
  - Any type/object meeting a distribution's needs is (in C++17) termed a URBG (Uniform Random Bit Generator).
  - A URBG + all the extra capabilities is known as an engine.

random_device is a URBG

- Designed to be a standard interface to any available environmental/physical source of randomness:
  - *E.g.*, /dev/random or /dev/urandom.
  - *E.g.*, a device that samples atmospheric noise.
- Its c'tor argument is an implementation-defined string.
- Its entropy( ) member estimates the device's entropy, defined as:

$$S = -\sum_{i=0}^{n-1} P_i \log P_i$$

Aside: entropy formula on Ludwig Boltzmann's tomb

— photo courtesy of Thomas Schneider

Physical sources of randomness

- “The Ferranti Mark I computer, first installed in 1951, …
  - “had a built-in instruction that put 20 random bits into the accumulator using a resistance noise generator; this feature had been recommended by A. M. Turing.” — Donald Knuth
- Hardware devices are commercially manufactured today:
  - Various hardware connections (USB, PCIe, …) are available.
  - Recent CPUs support RDRAND, RDSEED instructions.
- Free online services:
  - random.com, randomnumbers.info, fourmilab.ch/hotbits
  - Each seems amenable to a random_device interface.
  - But please take care, as there are also “psychic” services (offering lottery numbers and the like) for the unwary!

<random> provides 20 distribution types in 5 families

- 2 Uniform distributions:

  {uniform_int,
  uniform_real}_distribution

- 4 Bernoulli distributions:

  {bernoulli, binomial, geometric,
  negative_binomial}_distribution

- 5 Poisson distributions:

  {poisson, exponential,
  gamma, weibull,
  extreme_value}_distribution

- 6 Normal distributions:

  {normal, lognormal,
  chi_squared, cauchy, fisher_f,
  student_t}_distribution

- 3 Sampling distributions:

  {discrete, piecewise_constant,
  piecewise_linear}_distribution

? Why these dist's, not others?

  Early rationale in N1588: "On Random-Number Distributions for C++0x" (2004).

About std distributions

- Most of these are class templates whose template parameter is the desired type of the variate.
- Some deliver variates of only integer types, most others produce variates of only floating-point types:
  - *E.g.*, discrete_distribution< integral_type > .
  - *E.g.*, uniform_real_distribution< floating_point_type > .
  - Note: bernoulli_distribution gives only bool variates, so is a class (not a class template).
- While most std engines' results are bit-for-bit portable across platforms, distributions' results are not so specified:
  - Allows implementers to choose distribution algorithms (there are hundreds!) best for their target platforms.

Important features: interoperability & extensibility

- By design, any URBG can be used with any distribution.
- The standard carefully defines URBG and dist. interfaces:
  - Thus, users can provide their own URBGs or distributions…
  - And theirs will seamlessly interoperate with the std ones.
- Users have taken advantage of this flexibility:
  - New engine algorithms are providing this interface, *e.g.*, PCG (M. O'Neill, 2015) and Random123 (J. Salmon, *et al.,* 2011).
  - And gcc ships with numerous contributed distributions:
    - k_distribution
    - rice_distribution
    - beta_distribution
    - hoyt_distribution
    - pareto_distribution
    - arcsine_distribution
    - logistic_distribution
    - nakagami_distribution
    - triangular_distribution
    - von_mises_distribution
    - hypergeometric_distribution
    - uniform_on_sphere_distribution

Writing your own distribution?

- <random> provides a utility for experts:
  - template< class RealType, size_t bits, class URBG >
    RealType
    generate_canonical(URBG& g);
  - Calls g( ) as often as needed to get sufficient random bits, then maps those bits as uniformly as possible to the RealType range [0 .. 1).
- Often "a useful step in the process of transforming a value generated by a uniform random bit generator into a value that can be delivered by a random number distribution."

How to test a new distribution?

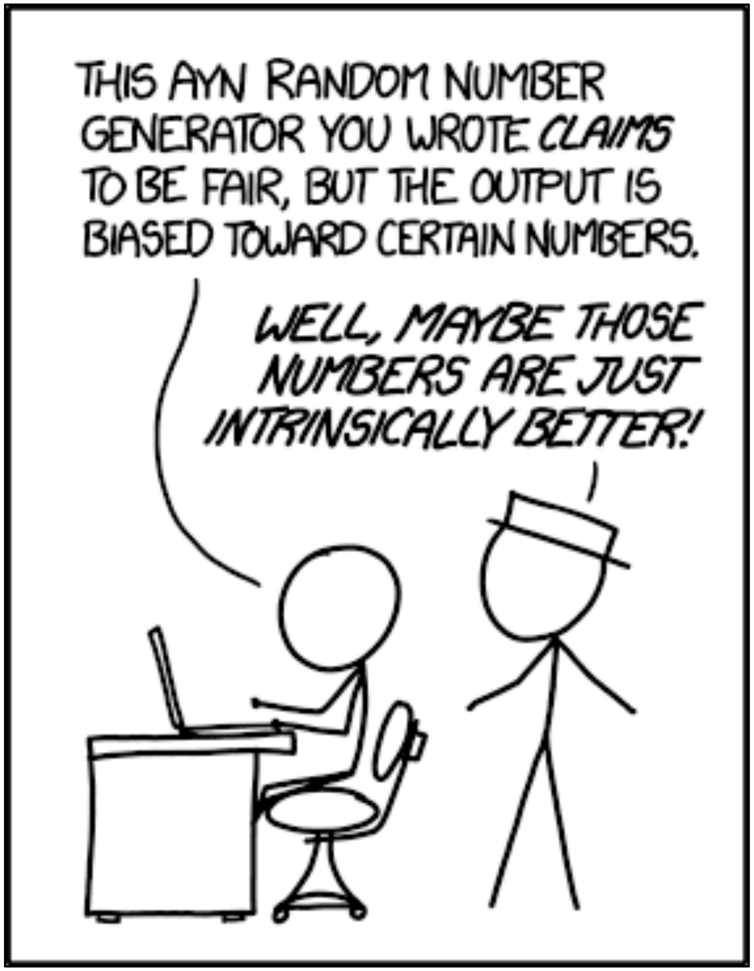
- Apply statistical tests:
  - Check the sample mean and sample variance (if applicable).
  - Consider χ^2 and/or Kolmogorov-Smirnov tests.
  - Failure? Try again with a different engine or engine seed.
  - Remember that "Statistical tests are *supposed* to fail occasionally, just not often!" — John D. Cook

☺ Ensure that "If the numbers are not random, they are at least higgledy-piggledy." — George Marsaglia

Abusing the <random> Header

> [Y]ou don't have to wait for someone to treat you bad[ly] repeatedly. All it takes is once, and if they get away with it that once … it sets the pattern for the future.
>
> — Jane Green

"Let not thy code introduce bias, lest ye die."



— Randall Munroe

"When it comes
to constructing [and using]
random number generators,
there is no such thing
as getting the software
almost right."

— Stephen K. Park
& Keith W. Miller

🛑 Don't do this! ⚠

❌ Example of the most common anti-pattern:

- Let's mimic the roll of a fair die:

  e( ) % 6 + 1 *// seems innocuous; what's wrong?*

① Assumes that e( ) can produce at least 6 values (*i.e.*, $2^n \geq 6$).

② Since 2^n isn't evenly divisible by 6, "this remainder operation makes lower numbers slightly more likely."

- "This is really awful!"
  - A floating-point (!) variation is "Hilariously non-uniform."
  - "It's the pigeonhole principle: if you have 2^{32} pigeons, … you can't put them into [6] pigeonholes without having more pigeons in some holes than others."

— Stephan T. Lavavej

Such a roll is unfair (biased)

- “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.
- “[T]o be correct, you must work with the distribution instead of destroy it.”

— Julienne Walker

- So let’s avoid such bigotry in programming, okay?

Recommended Header <random> Usage

Wisdom is knowing what to do next, skill is knowing how to do it, and virtue is doing it.

— David Starr Jordan

Let’s do better

- Analogy: If we wanted a variate v in [1 .. 5], we could roll a fair die, and re-roll (reject) if we get a 6:
  - int v;
    do v = roll_the_die( ); while(v == 6);
  - This is a simple example of a *rejection algorithm*.
- More generally, if we wanted v in [low .. high]:
  - int v;
    do v = e( ); while(v < low or v > high); *// naïve rejection*
  - But what if e’s range doesn’t include [low..high]? And what’s the performance if e’s range greatly exceeds [low..high]? *Etc.*
- We’re just re-inventing std::uniform_int_distribution< > :
  - Which is already part of <random>!

Advice: when you want random variates

- Instantiate an engine and a distribution.
- For each variate, call your distribution object …
- Passing it your engine object (as a source of randomness).
- Example: `auto variate = d(e);`
- Why? "It makes no sense to ask for a random number without some context. You need to specify random according to what distribution." — John D. Cook

Tip: manage engines and distributions as resources

- How many?
  - One of each is enough for many (most?) applications.
  - If so, can often treat them together:
    - *E.g.*, via pair, tuple, custom class, or class template.
    - *E.g.*, via bind, lambda, custom function, or function template.
- Engine objects are stateful and mutable, so:
  - Copy only when you want identical results in both places.
  - Usually pass by (non-const) reference, not by value.
  - Serialize access if using an engine in concurrent code:
    - Or consider thread_local engines (*i.e.*, 1 per thread).
- Ditto for distributions, so treat likewise:
  - *E.g.*, the *Box-Muller transform* calc's 2 normal variates at once.

Initializing (seeding) an engine

- An engine's initial state is influenced by providing a seed (starting value) to its c'tor:
  - std::default_random_engine e{ 13607 };   *// still reproducible*
  - A seed's type matches (or converts to) decltype(e( )), a.k.a. E::result_type (the engine's unsigned result type).
- Engine's size is a factor:
  - Can't initialize Mersenne twister well with a single integer.
  - std::seed_seq takes one integer and attempts to make more entropy, but not very successfully; probably best avoided.

Choosing good seeds can be challenging

- Reproducibility is critically important:
  - For some applications.
- But fatal for others:
  - “Players may be disappointed if your game always acts the same way given the same moves.” — John D. Cook
- Casting (to E::result_type) the current time or process id are common practices, but these are considered “low-quality seed data.”

What Else Can We Do with <random> ?

Can sample( ) a range (population)

- New algorithm in C++17:
  - template< class PopulationIter, class SampleIter
                   , class Size, class URBG >
    SampleIterator
       sample(PopulationIter first, PopulationIter last
                   , SampleIter out, Size wanted_size, URBG&& g);
  - Uses one of two different algorithms, depending on iterators' capabilities.
- An inplace_sample algorithm is being proposed; will partition rather than copy.

Can shuffle( ) a range (sequence)

- ```
  using  card_t  =  int;
  using  deck_t  =  array< card_t, 52 >
  deck_t  deck;   iota( begin(deck), end(deck), card_t{ } );
  ```
- ```
  using  engine_t  =  default_random_engine;     // vendor's choice
  using  seed_t     =  engine_t::result_type;
  engine_t   e { seed_t( time(0) ) };              //  a poor seed
  shuffle( begin(deck), end(deck), e );        // distribution is implicit
  ```
- ```
  auto  suit  =  [ ] ( card_t  c )  { return "♠♥♦♣" [ c / 13 ]; }
  auto  rank  =  [ ] ( card_t  c )  { return "A23456789TJQK" [ c % 13 ]; }
  auto  show  =  [ ] ( card_t  c )  { cout <<  ' '  <<  rank(c)  <<  suit(c); }
  for_each( begin(deck), end(deck), show );
  ```
- Note: the original STL random_shuffle algorithms (which predate <random>) were deprecated in C++14 and now removed in C++17.
  ```

Shuffling isn't always the right approach ☺

- ```
  template< class RA, class Engine >
  void
     shuffle_sort( RA b, RA e, Engine g )
  {
     while(  not  is_sorted(b, e)  )
        shuffle( b, e,   g );
  }
  ```
  ```

Distribution parameters can vary per call, if desired

```
template< class RA,  class URBG >
void  shuffle( RA const  b,  RA const  e,  URBG&&  g )
{
   using  diff_t      =  decltype( e − b );
   using  dist_t      =  uniform_int_distribution< diff_t >;
   using  param_t  =  typename  dist_t::param_type;  // takes c'tor args

   static  dist_t  d{ };    // distribution's range supplied per call, below
   // invariant: items [b..b+m) are shuffled; items [b+m..b+L] are unshuffled
   diff_t  const  L  =  e − b − 1;                      // b[L] is the last item
   for( diff_t  m{ };   m < L;   ++m )
      iter_swap( b + m                                  // the first unshuffled item
                     , b + d( g, param_t{m, L} )  // a random unshuffled item
                     );
}
```

Consider an iterator interface to <random>

- ```
  using  urbg_t       =  default_random_engine;
  using  variate_t    =  double;
  using  dist_t       =  uniform_real_distribution<variate_t>;
  ```
- ```
  urbg_t  g{ … };
  dist_t    d{ };  // default range is [ 0 .. 1 )
  variate_iterator<urbg_t, dist_t>  it{ g, d };   // see next page
  ```
- ```
  // create a random vector of extent N:
  vector<variate_t>  v( N );   copy_n( it, N, begin(v) );
  ```
- ```
  // dot product with a random vector:
  variate_t  prod{ inner_product( begin(v), end(v), it, 0.0 ) };
  ```
- ```
  // add noise to a vector:
  transform( begin(v), end(v)
           , begin(v)
           , [&] ( variate_t  d ) { return d  +  *it++; }  );
  ```
  ```

Sketch of an iterator interface to <random> 1

```
template< class URBG, class Dist >
class  variate_iterator
:  public  iterator< input_iterator_tag, typename Dist::result_type >
{
private:
   using  iter_t  =  variate_iterator;
   using  val_t   =  typename Dist::result_type;
   using  ptr_t   =  val_t const*;                  // returned by op –>
   using  ref_t   =  val_t const&;                  // returned by op *

   URBG*       u{ nullptr };     // non-owning
   Dist*       d{ nullptr };     // non-owning
   val_t       v{  };            // latest variate
   bool        valid{ false };
```

Sketch of an iterator interface to <random> 2

```
   // help:
   void   step(  ) noexcept  {  valid = false;  }
   ref_t  deref(  )  {  if( not valid )  v = (∗d)(∗u), valid = true;
                        return v;  }
public:
   // construct:
   constexpr  variate_iterator(  )  noexcept  =  default;
   variate_iterator( URBG& u, Dist& d )  :  u{ &u }, d{ &d }  {  ;  }
   // dereference:
   ref_t  operator ∗  (  )         {  return  deref( );  }
   ptr_t  operator –> (  )         {  return &deref( );  }
   // advance:
   iter_t&  operator ++ (  )       {  step( );  return ∗this;  }
   iter_t   operator ++ ( int )    {  iter_t  t{∗this};  step( );  return t;  }
}
```

Final <random> Thoughts

Caveat lector! 1

- WG21 published an early spec. of <random> in TR1 (2007):
  - To get feedback from a wider community of users.
  - Worked! The adopted version is much improved over that early version in many important ways.
  - Many of these improvements rely on newer C++11 language features, or are otherwise not backward-compatibile.
- But in the interim, early adopters commented on and offered advice regarding the then-new facility:
  - So the technical content of early writings (and, alas, of even some recent ones!) is often now at least somewhat suspect.
  - Even the historical content has often been wrong:
    - Boost.Random (J. Maurer) was our design's proof-of-concept!

Caveat lector! 2

- We’ve also found many <random>-related writings:
  - That give programming advice based on what seems to be a questionable understanding of the meaning, behavior, and properties of randomness, and/or …
  - That express opinion (commonly presented as fact) based on what seems to be a questionable understanding of <random>’s design.
- For all these reasons, I urge you:
  - Look carefully at such publications’ sources and dates, …
  - Before relying on their content.

Some resources of interest

- M. Haahr, random.org:
  - Brief tutorial re randomness.
  - Online environment-driven generation service.
- M. Klammler, P0205R0:
  - Recent proposal for enhanced (simplified) seeding; WG21 review under way.
- M. E. O'Neill, pcg-random.org:
  - New PCG engine description, code, and formal theory paper.
  - Blogs describing other usability improvements, serving as a basis for recent proposal …
  - P0347R0, by Song and O'Neill; WG21 review under way.

☺

<random> is really elegantly designed. I love this header.

— Stephan T. Lavavej

I think it's the best random number library design of all, by a mile. If I were a random number, I'd think I died and went to heaven.

— Andrei Alexandrescu

But not everyone agrees

- “To the extent that anyone cares about C++11's random-number facility at all, the C++ community is polarized between two views….
  - “It's amazingly elegant, a shining example of separation of concerns [with] a pluggable and extensible architecture [that’s] comprehensive and flexible.
  - “It's horrible to use, … unnecessarily overengineered …, completely unsuitable for beginners, and even seasoned programmers hate it.
- “Both camps are right.”

— M. E. O’Neill

- You’ll have to make up your own mind. Thanks very much.

What C++ Programmers Need to Know about Header <random>

FIN

WALTER E. BROWN, PH.D.

< webrown.cpp @ gmail.com >