Three **<random>**-related Proposals

Document #: WG21 N3547
Date: 2013-03-12
Revises: None
Project: JTC1.22.32 Programming Language C++
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

This paper proposes (1) to add one function template to **<algorithm>**, (2) to add a few novice-friendly functions to **<random>**, and (3) to deprecate some related legacy interfaces. The unifying factor in this tripartite proposal is the entities' respective connection with random numbers.

1 Additions to <algorithm>

1.1 Proposal

In [Aus08a], Matt Austern proposed a number of additional algorithms for the standard library. Among these were **random_sample** and **random_sample_n**, based respectively on the well-known algorithms S ("selection sampling technique") and R ("reservoir sampling") elucidated in [Knu97, §3.4.2]. The paper succinctly describes these algorithms as "... two important versions of random sampling, one of which randomly chooses n elements from a range of N elements and the other of which randomly chooses n elements from an input range whose size is initially unknown. ..."

After WG21 consideration at the Sophia-Antipolis meeting, Austern updated the proposal. Among other changes, he withdrew these two algorithms: "The LWG was concerned that they might not be well enough understood for standardization. ... It may be appropriate to propose those algorithms for TR2" [Aus08b]. The wiki minutes of the discussion are equally terse: "Bjarne feels rationale is insufficient. PJ worries we will get it wrong. Lawrence worries that people will roll their own and get it wrong. Good candidate for TR2" [LWG08]. The subsequent vote regarding these proposed algorithms achieved a solid LWG consensus (10-1, 2 abs.) in favor of their future inclusion in a Technical Report (now termed a Technical Specification).

Since WG21 is now preparing updates to C++11 and drafting new Technical Specifications, it is an appropriate time to reconsider adding these algorithms, which (as Austern pointed out) have long been part of SGI's STL implementation. Unlike the SGI implementation on which the previous proposal was based, the present proposal features a unified (common) interface, **sample**, that selects between algorithms R and S based on the categories of the iterators supplied at the point of call.

1.2 Expository implementation[1]

First, here is the common interface. It relies on overload resolution with tag dispatch to select the appropriate algorithm based on its arguments' iterator categories:

```
template< class PopIter, class SampleIter, class Size, class URNG >
inline SampleIter
  sample( PopIter first, PopIter last
        , SampleIter out
        , Size n, URNG && g
        )
{
  using pop_t  = typename iterator_traits<PopIter   >::iterator_category;
  using samp_t = typename iterator_traits<SampleIter>::iterator_category;

  return _sample( first, last, pop_t{}
                , out, samp_t{}
                , n, std::forward<URNG>(g)
                );
}
```

We next exhibit the reservoir sampling algorithm R.[2] It requires only input interators for access to the population being sampled, but needs a random access iterator to (the start of) the reservoir that will ultimately hold the resulting sample:

```
template< class PopIter, class SampleIter, class Size, class URNG >
SampleIter
  _sample( PopIter first, PopIter last, input_iterator_tag
         , SampleIter out, random_access_iterator_tag
         , Size n, URNG && g
         )
{
  using dist_t = uniform_int_distribution<Size>;
  dist_t d{};

  Size sample_sz{0};
  while( first != last && sample_sz != n )
    out[sample_sz++] = *first++;

  for( Size pop_sz{sample_sz};  first != last;  ++first )  {
    using param_t = typename dist_t::param_type;
    param_t const p{0, ++pop_sz};
    Size k{ d(g, p) };
    if( k < n )
      out[k] = *first;
  }
  return out + sample_sz;
}
```

We conclude this section with the selection sampling algorithm S. It needs forward iterators to access the population, but only an output iterator to the resulting sample:

---

[1] Beware of bugs in this code; I have only tried it, not proved it correct. [☺ Apologies to Knuth.]

[2] Knuth's original algorithm R guarantees stability, but at the cost of an extra level of indirection, an additional sorting step, etc. The SGI version, as proposed here and in Austern's earlier paper, sacrifices stability in the interest of performance, but retains all other characteristics, especially the all-important randomness of the resulting sample.

```
template< class PopIter, class SampleIter, class Size, class URNG >
SampleIter
  _sample( PopIter first, PopIter last, forward_iterator_tag
         , SampleIter out, output_iterator_tag
         , Size n, URNG && g
         )
{
  using dist_t = uniform_int_distribution<Size>;
  dist_t d{};

  Size unsampled_sz = distance(first, last);
  for( n = min(n, unsampled_sz);  n != 0;  ++first, --unsampled_sz )  {
    using param_t = typename dist_t::param_type;
    param_t const p{0, unsampled_sz};
    if( d(g, p) < n )
      *out++ = *first, --n;
  }
  return out;
}
```

2 Additions to <random>

2.1 Proposal

We propose to add to **<random>** the following modest 4-part toolkit of novice-friendly functions:[3]

- **global_urng()**
  Grants access to a URNG object of implementation-specified type.
- **randomize()**
  Sets the above URNG object to an (ideally) unpredictable state.
- **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 further propose to give the existing algorithm **shuffle** and the above-proposed algorithm **sample** a default argument for their respective parameters of type **UniformRandomNumberGenerator &&**.[4] The proposed default value is the result of calling the above-proposed **global_urng()**.

2.2 Representative implementation

Here, first, is a sample implementation of the **global_urng()** and **randomize()** functions, demonstrating an implementor's choice of an engine type for the shared URNG:

[3] This proposal is in response to the near-incessant user ~~whining~~ ~~kvetching~~ feedback, from even respected C++ *cognoscenti*, that **<random>**'s contents are insufficiently "friendly" to novices and other casual users.

[4] Neither of these parameters currently specifies any default argument.

```
using __urng_t = std::default_random_engine;
__urng_t &  global_urng( )
{
  static __urng_t  u{};
  return u;
}

void  randomize( )
{
  static std::random_device  rd{};
  global_urng().seed( rd() );
}
```

If the implementor chooses differently, the same functions might be implemented this way:

```
using __urng_t = std::random_device;
__urng_t &  global_urng( )
{
  static __urng_t  rd{};
  return u;
}

void  randomize( )  { }
```

In contrast, the **pick_a_number** functions can be written so as to be agnostic with respect to the implementor's choice of URNG type:

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

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

3 Deprecations in <cstdlib> and <stdlib.h>

By common consensus at several consecutive WG21 meetings during which the C++11 random number facility was being discussed and shaped into its final form, it has for a number of years been the long-term plan to excise the legacy C random number facility from the **std** namespace. Indeed, obliquely acknowledging the quality[5,6] of C++11's **<random>** header, WG21 voted several years ago to insert a Note[7] into [c.math]/5 as a head start on this plan: "The random number generation . . . facilities in this standard are often preferable to **rand**."

[5] "[B]y and large, 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" [Ale07].

[6] "The C++11 <random> is very STL-like in that it sets up requirements for random number generators. . . , and random distributions. . . , and then the client can mix and match the two. It's a really very cool design [Hin12].

[7] The language for this Note was proposed in [Daw08]; [Bec08] was the first Working Paper to incorporate it.

In light of our concurrent proposal, above, for a more user-friendly interface to **<random>**, we therefore propose to execute the next step of the plan to discourage the use of the traditional C function **rand()** as well as its associated seeding function **srand()** and upper limit macro **RAND_MAX**. (These are declared in the traditional C header **<stdlib.h>** and the corresponding C++ header **<cstdlib>**.) We propose to begin this transition by formally deprecating:

- **std::rand()**, **std::srand()**, and (when obtained via **<cstdlib>**) **RAND_MAX** and
- algorithm **random_shuffle()** (keeping **shuffle**, however).

4 Proposed wording

All proposed wording is relative to WG21 draft [DuT12]. It is recommended to apply the wording additions (marked in green text) and deletions (marked in red, usually ~~struck~~ as well) in the order shown. Editorial notes are displayed against a gray background.

Append the following new paragraph to [depr.c.headers]:

Use of function **rand**, function **srand**, and macro **RAND_MAX** is deprecated when these names are introduced via the **<cstdlib>** header. Use of function **std::rand** and function **std::srand** is deprecated when these names are introduced into the **std** namespace via the header **<stdlib.h>**.

Copy all of the current [alg.random.shuffle] to a new section in Annex D, applying the changes shown below.

Random shuffle [depr.alg.random.shuffle]

The following templates are in addition to those specified in Clause [alg.random].

```
template<class RandomAccessIterator>
  void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class RandomNumberGenerator>
  void random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
                      RandomNumberGenerator&& rand);

template<class RandomAccessIterator, class UniformRandomNumberGenerator>
  void shuffle(RandomAccessIterator first, RandomAccessIterator last,
               UniformRandomNumberGenerator&& g);
```

Effects: Permutes the elements in the range [**first**, **last**) such that each possible permutation of those elements has equal probability of appearance.

Requires: **RandomAccessIterator** shall satisfy the requirements of **ValueSwappable** (17.6.3.2). The random number generating function object **rand** shall have a return type that is convertible to **iterator_traits<RandomAccessIterator>::difference_type**, and the call **rand(n)** shall return a randomly chosen value in the interval [0, **n**), for **n** > 0 of type **iterator_traits<RandomAccessIterator>::difference_type**. ~~The type `UniformRandomNumberGenerator` shall meet the requirements of a uniform random number generator (26.5.1.3) type whose return type is convertible to `iterator_traits<RandomAccessIterator>::difference_type`.~~

Complexity: Exactly (**last** − **first**) − 1 swaps.

Remarks: To the extent that the implementation of these functions makes use of random numbers, the implementation shall use the following sources of randomness:

The underlying source of random numbers for the first form of the function is implementation-defined. An implementation may use the **rand** function from the standard C library.

In the second form of the function, the function object **rand** shall serve as the implementation's source of randomness.

~~In the third `shuffle` form of the function, the object `g` shall serve as the implementation's source of randomness.~~

In the synopsis in [algorithms.general], (a) remove both overloads of **random_shuffle**, and (b) insert = global_urng( ) as the default value for the last parameter of **shuffle**.

Change the heading of [alg.random.shuffle] as follows and then make the remaining indicated adjustments to the retitled section's text:

25.3.12 Random shuffleing and sampling [alg.random.~~shuffle~~]

```
~~template<class RandomAccessIterator>~~
~~  void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);~~

~~template<class RandomAccessIterator, class RandomNumberGenerator>~~
~~  void random_shuffle(RandomAccessIterator first, RandomAccessIterator last,~~
~~                       RandomNumberGenerator&& rand);~~

template<class RandomAccessIterator, class UniformRandomNumberGenerator>
  void shuffle(RandomAccessIterator first, RandomAccessIterator last,
               UniformRandomNumberGenerator&& g=global_urng());
```

Effects: Permutes the elements in the range [**first**, **last**) such that each possible permutation of those elements has equal probability of appearance.

Requires: **RandomAccessIterator** shall satisfy the requirements of **ValueSwappable** (17.6.3.2). ~~The random number generating function object `rand` shall have a return type that is convertible to `iterator_traits<RandomAccessIterator>::difference_type`, and the call `rand(n)` shall return a randomly chosen value in the interval~~ [0, **n**)~~, for n > 0 of type `iterator_traits<RandomAccessIterator>::difference_type`.~~ The type **UniformRandomNumberGenerator** shall meet the requirements of a uniform random number generator (26.5.1.3) type whose return type is convertible to **iterator_traits<RandomAccessIterator>::difference_type**.

Complexity: Exactly (**last** − **first**) − 1 swaps.

Remarks: To the extent that the implementation of ~~these~~ this function~~s~~ makes use of random numbers, ~~the implementation shall use the following sources of randomness:~~

~~The underlying source of random numbers for the first form of the function is implementation-defined. An implementation may use the `rand` function from the standard C library.~~

~~In the second form of the function, the function object `rand` shall serve as the implementation's source of randomness.~~

~~In the third `shuffle` of the function,~~ the object **g** shall serve as the implementation's source of randomness.

Append the following text to the newly retitled [alg.random], also incorporating the signature into the header **<algorithm>** synopsis at the beginning of [algorithms].

```
template <class PopulationIterator, class SampleIterator,
          class Distance, class URNG>
  SampleIterator sample(PopulationIterator first, PopulationIterator last,
                        SampleIterator out, Distance n,
                        UniformRandomNumberGenerator&& g=global_urng());
```

Requires:

- **PopulationIterator** shall meet the requirements of an **InputIterator** type.
- If **PopulationIterator** meets the additional requirements of a **ForwardIterator** type, **SampleIterator** shall meet the requirements of an **OutputIterator** type. Otherwise, **SampleIterator** shall meet the requirements of a **RandomAccessIterator** type.
- **PopulationIterator**'s value type shall be **Assignable**, and shall be writable to **out**.
- **Distance** shall be an integer type.
- **UniformRandomNumberGenerator** shall meet the requirements of a uniform random number generator ([rand.req.urng]) type whose return type is convertible to **Distance**.
- **out** shall not be in the range [**first**, **last**).

Effects: Copies $min($**last**$-$**first**, **n**$)$ elements (the *sample*) from [**first**, **last**) (the *population*) to **out** such that each possible sample has equal probability of appearance.

Returns: The end of the resulting sample range.

Complexity: No worse than $\mathcal{O}($**n**$)$.

Remarks:

- Stable if and only if **PopulationIterator** meets the requirements of a **ForwardIterator**.
- To the extent that the implementation of this function makes use of random numbers, the object **g** shall serve as the implementation's source of randomness.

Append the following text as a new subsection under [rand.util]:

Global URNG functions [rand.util.global]

```
see below& global_urng();
```

Effects: Ensures the instantiation of an object u that has static storage duration and whose implementation-specified type meets the requirements of a uniform random number generator [rand.req.urng] type. [*Note:* The implementation may select this type on the basis of performance, size, quality, or any combination of such factors, so as to provide at least acceptable behavior for relatively casual, inexpert, and/or lightweight use. — *end note*] The extent to which u's initial state can be predicted or can ever be reproduced is unspecified.

Synchronization: It is implementation-defined whether this function may introduce data races [res.on.data.races].

Returns: An lvalue reference to u.

```
void randomize();
```

Effects: Sets the uniform random number generator given by **global_urng()** to an (ideally) unpredictable state.

```
int pick_a_number(int from, int thru);
double pick_a_number(double from, double upto);
```

Returns: Returns a variate uniformly distributed (a) in the closed range [**from**, **thru**], for the first form of the function, and (b) in the half-open range [**from**, **upto**), for the second form of the function.

Remarks: To the extent that the implementation of these functions makes use of random numbers, the implementation shall use, as its source of randomness, the uniform random number generator given by **global_urng()**.

5 Summary and conclusion

This paper has proposed three modest changes to the C++11 standard, each related to the random number facility. The proposals are largely independent of each other, but we respectfully recommend their adoption together via approval of the paper in its entirety.

6 Acknowledgments

Many thanks to the readers of early drafts of this paper for their thoughtful comments, and to Matt Austern for the original proposal of the sampling algorithms.

7 Bibliography

[Ale07] Andrei Alexandrescu: "Re: Conveniently generating random numbers with TR1 random." comp.std.c++, 2007-06-11.

[Aus08a] Matt Austern: "More STL algorithms." ISO/IEC JTC1/SC22/WG21 document N2569 ((mailing)post-Bellevue mailing), 2008-02-29.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2569.pdf.

[Aus08b] Matt Austern: "More STL algorithms (revision 2)." ISO/IEC JTC1/SC22/WG21 document N2666 ((mailing)post-Sophia mailing), 2008-06-11.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2666.pdf.

[Bec08] Pete Becker: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N2691 ((mailing)post-Sophia mailing), 2008-06-27.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2691.pdf.

[Daw08] Beman Dawes et al.: "Thread-Safety in the Standard Library (Rev 2)." ISO/IEC JTC1/SC22/WG21 document N2669 ((mailing)post-Sophia mailing), 2008-06-13.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2669.pdf.

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

[Hin12] Howard Hinnant: Untitled response to posted query. 2012-31-07.
http://stackoverflow.com/questions/11717433/tutorial-or-example-code-for-extending-c11-random-with-generators-and-distribu.

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

[LWG08] "Library Working Group: Wednesday." Minutes of LWG meeting, Sophia-Antipolis, France. 2008-06.
http://wiki.edg.com/twiki/bin/view/Wg21sophiaAntipolis/LibraryWorkingGroup.

8 Revision history

Revision	Date	Changes
1.0	2013-03-12	• Published as N3547.