A **sample** Proposal, v4

Document #: WG21 N3925
Date: 2014-02-14
Revises: N3842
Project: JTC1.22.32 Programming Language C++
Reply to: Walter E. Brown <webrown.cpp@gmail.com>

Contents

Abstract

This paper proposes to add to the standard library an interface for two algorithms carrying out random sampling. Such algorithms have long been part of the original (SGI) implementation of the Standard Template Library.

[S]ampling is the first step if you are looking to move things forward.
— Tom Pirko

1 Proposal

In [N2569], Matt Austern proposed several additional algorithms (mostly taken from SGI's original STL implementation[1]) 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 by Donald Knuth in [Knu97, §3.4.2]. Austern's 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, producing [N2666]. Among other changes, he withdrew the sampling algorithms because "The LWG was concerned that they might not be well enough understood for standardization. . . . It may be appropriate to propose those algorithms for TR2." LWG subsequently achieved a solid consensus (10-1, 2 abs.) in support of having these algorithms in a future Technical Report (now termed a Technical Specification).

Given WG21's stated near-term intent to publish such documents, and thereafter to issue a revised standard, we believe it is an appropriate time to reconsider these algorithms for the C++ standard library. Unlike the SGI implementation on which Austern's 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] See http://www.sgi.com/tech/stl.

2 Expository implementation[2]

Common interface As an implementation detail, we name both sampling algorithms **`__sample`**. The proposed **`sample`** template then relies on overload resolution via tag dispatch to choose between those algorithms:

```
template< class PopIter, class SampleIter, class Size, class URNG >
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, forward<URNG>(g)
                 );
}
```

Reservoir sampling Our proposed version of the reservoir sampling algorithm R[3] requires only input iterators 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>;
  using param_t = typename dist_t::param_type;
  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, ++pop_sz )  {
    param_t const p{0, pop_sz};
    Size const k{ d(g, p) };
    if( k < n )  out[k] = *first;
  }
  return out + sample_sz;
}
```

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

[3]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.

Selection sampling Our proposed selection sampling algorithm S needs forward iterators to access the population, but only an output iterator to 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>;
  using param_t = typename dist_t::param_type;
  dist_t d{};

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

3 Proposed wording[4]

Change the heading of [alg.random.shuffle] as shown:

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

Append the following text to the newly retitled [alg.random.shuffle], also incorporating the new declaration into **<algorithm>**'s synopsis at the beginning of [algorithms].

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

Requires:

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

[4] All proposed additions and ~~deletions~~ are relative to the post-Chicago Working Draft [N3797]. Editorial notes are displayed against a gray background.

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. [*Note:* Algorithms that obtain such effects include *selection sampling* and *reservoir sampling*. *— end note*]

Returns: The end of the resulting sample range.

Complexity: $\mathcal{O}($**`n`**$)$.

Remarks:

- Stable if and only if **`PopulationIterator`** meets the requirements of a **`ForwardIterator`** type.
- 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.

4 Feature-testing macro

For the purposes of SG10, we recommend the macro name **`__cpp_lib_sample`**.

5 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.

6 Bibliography

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

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

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

[N3797] Stefanus Du Toit: "Working Draft, Standard for Programming Language C++." ISO/IEC JTC1/SC22/WG21 document N3797 (post-Chicago mailing), 2013-10-13. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf.

7 Revision history

Version	Date	Changes
1	2013-03-12	• Published as N3547.
2	2013-08-30	• Corrected punctuation of WG21 bib entries. • Relocated **`++pop_sz`** within reservoir algorithm. • Relocated **`--unsampled_sz`** within selection algorithm. • Tweaked fonts and spacing. • Added epigraphs. • Recoded to use C++14 features. • Published as N3742.
3	2014-01-01	• Retitled so as to focus on only the sampling algorithm proposal. • Discarded everything not relevant to sampling. • Fixed a few typos and clarified some wording. • Tweaked the expository code. • Added feature-testing macro recommendation. • Published as N3842.
4	2014-02-14	• Tweaked proposed wording per LWG guidance. • Published as N3925.