Improvements to TR1's Facility for Random Number Generation

Document #: WG21/N1933 = J16/06-0003
Date: 2006-02-23
Revises: None
Project: Programming Language C++
Reference: ISO/IEC IS 14882:2003(E)
Reply to: Walter E. Brown<wb@fnal.gov>
Mark Fischler <mf@fnal.gov>
Jim Kowalkowski <jbk@fnal.gov>
Marc Paterno <paterno@fnal.gov>
CEPA Dept., Computing Division
Fermi National Accelerator Laboratory
Batavia, IL 60510-0500
U.S.A

Contents

If the numbers are not random, they are at least higgledy-piggledy.

— GEORGE MARSAGLIA

I cannot do it without compters.

— WILLIAM SHAKESPEARE

1 Introduction

This document is intended to accompany [BFKP06], our proposal that the C++0X standard library incorporate a facility for random number generation. The present paper describes the process by which we arrived at that proposal, and provides explanation and rationale for our major decisions along the way.

2 A brief history

Based on Maurer's updated proposal [Mau03], TR1 [Aus05] specified a facility for the generation of random numbers, introducing to C++ the concepts of uniform random number generator, [random number] engine, and [random number] distribution, among others. We were and remain strong supporters of this facility, believing it to be a standard library component important to a significant number of user communities.

However, we quickly recognized that the facility as articulated in TR1 could be profitably improved. In particular, [Pat04] argued for improvements in the criteria to determine the set of random number distributions to be incorporated, along the way identifying five important families[1] of distributions to be supported. These revised criteria received favorable review and subsequent approval by the Library Working Group. Detailed wording for the additional distributions meeting these newly-accepted criteria was provided in [PFBK05].

Most recently, we produced the companion paper [BFKP06]. Hereinafter known as the Proposal, that paper began as an edited amalgamation of TR1's clause 5.1 ("Random number generation") with the new distributions' wording as provided in [PFBK05]. We then looked at the relevant unresolved issues from [Hin05], and incorporated proposed resolutions. We also addressed the issues raised on the reflector by Austern on 2006-01-10 per the responses by Dos Reis, Maurer, and Myers. Finally, we incorporated a few new elements (and removed an old one or two) based on our experiences in implementing the TR1 faciltiies and supplementary distributions.

The remainder of the present paper will describe in more detail the considerations and decisions made in arriving at our current proposal. The order of topics will roughly correspond to their order in the Proposal. We begin with an overview of the structure of the Proposal, and then discuss our changes and additions to the Requirements section.

3 General structure

The general structure of the Proposal for "Random number generation" is fundamentally identical to that of TR1's clause 5.1: an introduction followed by subsections providing the details. We have taken the liberty of proposing a specific context for this facility's future incorporation into the Working Paper, namely that of clause 26 ("Numerical facilities"). The introductory material has been expanded with definitions of general utility throughout the subclause. Some of these definitions were moved from elsewhere in the TR1 version; a few definitions introduce new descriptive nomenclature that permitted subsequent specifications to be simplified.

There are only two major changes at the subsection level: The subsection specifying the `variate_generator` component has been eliminated, and the subsection specifying random number engines has been split into two in order to distinguish between *random number engines* and *random number engine adaptors*. Each of these decisions is separately discussed below.

At the next level, we have generally arranged items (*e.g.*, the engines) in alphabetical order to make them easier to locate within the text. Finally, we organized the distributions according to the principles set forth in [PFBK05], giving each distribution family its own subsubsection, with its distributions alphabetically presented therein.

4 New and changed requirements

The Proposal introduces a new "General requirements" subsection, obtained largely by factoring common requirements from elsewhere in the TR1 wording. We believe that requirements

[1] These families of distributions were termed the *uniform*, *Bernoulli*, *Poisson*, *normal*, and *sampling* families.

and definitions that are common to multiple concepts are best factored out from those specific concepts.

4.1 Uniform random number generator requirements

The Proposal makes one important change to the URNG requirements. In TR1, member functions `min()` and `max()` were required of a URNG. In the Proposal, the corresponding members `min` and `max` (the *extrema*) have become compile-time constants. We arrived at this decision by considering separately the cases of integer-valued and real-valued URNGs.

For integer-valued URNGs, in all cases the appropriate values will be of the type denoted by `result_type`. Further, in all cases the correct values are known as a consequence of specifying the URNG's algorithm, and in each case are computable at compile-time.

For real-valued URNGs, we selected `int` as the extrema's type, and fixed their values at 0 and 1, respectively. We did so because real-valued engines either use modular floating arithmetic, where the natural modulus is 0.0 to 1.0, or else they scale an integer engine, invariably yielding a value in the open interval $(0.0, 1.0)$[2].

The real gain of the change in the URNG requirement is in the simplification of a distribution's use of engines' results. It is also an important factor leading to our decision (discussed below) to eliminate `variate_generator`. Finally, each of the URNGs individually described in the subsequent engine and engine adaptor subsections has been updated by specifying the correct values of their extrema.

4.2 Random number engine requirements

The Proposal's main addition to the requirements for a random number engine is the introduction of the nomenclature *transition algorithm* and *generation algorithm*. This permitted us to simplify virtually all of the specifications of the individual engines later in the subclause.

The Proposal also provides updated wording specifying the behavior of `operator==`. It now more precisely reflects the intended semantics.

4.3 Random number engine adaptor requirements

The Proposal formally introduces the notion of a *random number engine adaptor*. All the elements of this concept were already present in TR1, but were not singled out under a common umbrella. By doing so, we identify a valuable concept, and provide substantially better guidance for users who may wish to define an *adaptor* of their own.

While every *random number engine adaptor* must meet the requirements of a *random number engine*, the converse is not true. It therefore seemed to us desirable to separate these notions, and to identify explicitly the additional behaviors and interpretations to which an *adaptor* is subject.

The requirements articulated in this new subsection are merely a centralization and formalization of the corresponding requirements and understandings present in TR1. No code changes are required.

---

[2] The argument to exclude either or both endpoints is that mathematically the probability of hitting the endpoint approaches 0, and it is more efficient if we can avoid checks for the special cases of endpoints (*e.g.*, if we take $\log x$, we want 0 excluded; if we take $1/(1-x)$, we want 1 excluded).

The argument to include either or both endpoints is that it may be more efficient (or "natural") for a URNG to produce a range that includes an endpoint.

We prefer to mandate the exclusion of both endpoints because this is easily accomplished with at most two additional arithmetic operations in cost (namely, multiplying by $1-\varepsilon$ and adding any value less than $\varepsilon/2$), while the cost of not excluding the endpoints is much more expensive for the user: the user must code some conditional or else must do a more general set of operations than might be needed for any specific URNG.

4.4 Random number distribution requirements

The Proposal encompasses three changes to the requirements for a random number distribution. One is a deletion, the second is an addition, and the third makes a uniform requirement out of an *ad hoc* one.

4.4.1 No `input_type` requirement

The Proposal deletes the requirement that a distribution contain a nested `input_type`, thereby instead (implicitly) requiring that a distribution accept any URNG, independent of the URNG's `result_type`. Our implementation experience has shown that this is not difficult to achieve. Further, it avoids the artificial type proliferation introduced by restricting a distribution to work with only certain kinds of URNGs.

4.4.2 New `param_type` requirements

The Proposal adds requirements centered on the notion of a distribution's `param_type`, an implementation-defined type corresponding to the distribution's parameter(s), and that is constructible from the identical values of the parameters used in the construction of the distribution. Accompanying the new nested `typedef` are the following new distribution member functions:

1. A constructor to initialize the distribution from an instance of the `param_type`,
2. An accessor that obtains a `param_type` instance corresponding to the distribution's parameters,
3. A mutator that changes the distribution's parameters so as to correspond to those of its `param_type` argument, and
4. A "one-shot" `operator()` overload that produces a random variate as if the distribution had been constructed according to the values of its `param_type` argument. (This is not completely new, since TR1's `variate_generator` provided such functionality, although it was not usable in connection with all distributions.)

These functions provide a uniform interface so that distribution parameters can be manipulated generically: it now becomes possible to write code that is independent of any specific distribution.

4.4.3 Consistent `min()` and `max()` requirements

Observing that many, but not all, TR1 distributions provide `min()` and `max()` member functions, and noting that all mathematical distributions have such extrema, the Proposal requires these functions of all distributions. The values returned are specified so as to reflect properties of the underlying mathematical distribution, and not any quirks of any implementation.

The Proposal further requires that the returned values take into account the current state of the distribution object. That is why, unlike URNGs, these members must be functions rather than `static const` data members.

Table 1 provides hints to implementors regarding the values to be returned by these functions. In that table, ∞ denotes the value `numeric_limits<result_type>::infinity()` if `numeric_limits<result_type>::has_infinity()` is true; otherwise it denotes the value `numeric_limits<result_type>::max()`.

..._distribution	min()	max()
uniform_int	a	b
uniform_real	a	b
bernoulli	true if p is 1; otherwise false	false if p is 0; otherwise true
binomial	t if p is 1; otherwise 0	0 if p is 0; otherwise t
geometric	0	∞
negative_binomial	0	∞
poisson	0	∞
exponential	0	∞
gamma	0	∞
weibull	0	∞
extreme_value	$-\infty$	∞
normal	$-\infty$	∞
lognormal	0	∞
chi_squared	0	∞
cauchy	$-\infty$	∞
fisher_f	0	∞
student_t	$-\infty$	∞
discrete	0	$n-1$
piecewise_constant	the lower edge of the first bin with non-zero weight	the upper edge of the last bin with non-zero weight
general_pdf	the smallest value of x for which pdf(x) is non-zero	the largest value of x for which pdf(x) is non-zero

Table 1: Results of distributions' min() and max() functions

5 Header <random> synopsis

The entries in the Proposal's synopsis have been reordered to correspond to the order in which their respective formal descriptions are provided in the subsequent text. Additionally, the items from the "Engines with predefined parameters" section, absent from the TR1 synopsis, are synopsized in the Proposal in order to produce a complete overview of all the random number generation components.

6 The demise of variate_generator

In TR1, variate_generator is useful because TR1's real-valued URNGs are not required to be *normalized* (produce values in the range $(0, 1)$), and because TR1's integer-valued URNGs are not required to have compile-time min and max. Therefore, variate_generator provides a standard way to avoid logically unnecessary computation during each invocation of a distribution. Moreover, since each TR1 distribution has its own input_type, variate_generator is useful in order to provide a standard way of tying an arbitrary URNG to an arbitrary distribution such that the distribution receives its correct input_type.

As a result of our implementation experience, we have found that it is easier to make each distribution cope with arbitrary input_types than it is to decide on the "best" input_type corresponding to each distribution. Further, after careful perusal of the URNGs specified in TR1, every one of them either (a) already met our definition of normalized or (b) already had min and max values that could have been expressed at compile-time. Therefore, it seems the utility of the

`variate_generator` to provide preparation for more efficient calls to `operator()` is no longer necessary.

It has also been our experience that there is a need for additional modes of sharing (or not sharing) engines and/or distributions that are not supported by the TR1 design of `variate_generator`. It is not clear that any single design can accommodate all the useful variations in this regard.

The remaining utility of the `variate_generator` is to provide a niladic function that a user can conveniently call. This is a small syntactic convenience, since absent the `variate_generator`, the user would merely call `d(e)` directly. We note that if such a niladic function object is truly required by the user, one can be coded in a straightforward fashion via TR1's `bind` and `function` facilities[3]. Here is a comparison of the two approaches (namespaces omitted for clarity):

```
1  //                                Listing 1
2  variate_generator< minstd_rand0
3                   , uniform_real_distribution<double>
4                   >
5    vg1( minstd_rand0()
6       , uniform_real_distribution<double>(-10.0, +10.0)
7       );
8
9  function< double() >
10   vg2( bind( uniform_real_distribution<double>(-10.0, +10.0)
11            , minstd_rand0()
12      )     );
```

Further, it is not difficult to provide a class template for function objects that simultaneously support both niladic and unary calls:

```
1  //                                Listing 2
2  template< class Distribution, class Engine >
3  class VG  {
4  public:
5    typedef  typename Distribution::result_type  result_type;
6    typedef  typename Distribution::param_type   param_type;
7
8    VG( Distribution & d, Engine & e )
9      : f0( std::tr1::bind(d, e    ) )
10     , f1( std::tr1::bind(d, e, _1) )
11   { }
12
13   result_type  operator() (                       )  { return f0(); }
14   result_type  operator() ( param_type const & p )  { return f1(p); }
15
16 private:
17   std::tr1::function< result_type (                  ) >  f0;
18   std::tr1::function< result_type (param_type const &) >  f1;
19 };
```

For the above reasons, with no loss of generality and with no loss of functionality, we decided to omit `variate_generator` from the Proposal.

[3] While it is our firm hope that these and the other TR1 *function objects* [tr.func] will be incorporated into the Working Paper, such incorporation is not part of the Proposal.

7 Random number engine adaptor class templates

The Proposal includes this new subsection, corresponding to the new requirements described above for *random number engine adaptor*. The descriptions of the `discard_block` and `xor_combine` templates were moved here from the "Random number engine class templates" subsection, consistent with their identification as engine adaptors rather than merely engines.

Having identified the underlying engine adaptor concept, we felt it appropriate to revisit the early decision (in [Mau03]) regarding which engines a random number generation facility should support. With our new understanding of engine adaptors, we reasoned that there might be algorithms previously considered purely as engines that now might be worthy of consideration as adaptors. We indeed discovered such a (well-known) algorithm, and included it in the Proposal under the name `shuffle_order_engine`.

This is Knuth's reordering algorithm B [Knu98, p. 34], credited to [BD76]. Superficially, this algorithm appears similar to that described in Clause 25 under the name `random_shuffle`: both do rearrange values of some underlying sequence. However, the `shuffle_order_engine` algorithm has been demonstrated to have good randomness properties while the `random_shuffle` algorithm, if used as an adaptor to an underlying engine, has no such known properties.

8 Engines with predefined parameters

The Proposal restructures the text of this subclause, and adds one new predefined engine based on the new `shuffle_order_engine`. The Proposal also removes all *unspecified* types from the template arguments, substituting `unsigned long`; we saw no reason to grant implementation latitude (of dubious value in this context) at the expense of user stability.

Finally, there is an important correction to the definition of `ranlux64_base_01`.

9 Random number distribution class templates

The Proposal employs `int` as the type for expressing all integral distribution parameters, and employs `double` as the type for expressing all floating-point distribution parameters. This is, in part, a consequence of the decision (described above) to remove all `input_type` template parameters, since constructors can no longer rely on this user-specified type. However, there is little or no loss of generality, as one would have to produce a truly staggering number of variates in order to detect the difference between a distribution constructed with a `double` and one constructed with a `long double` parameter.

The Proposal's other changes in this section were previously mentioned: a new organization per [PFBK05]'s categorizations, and incorporation of the additional distributions there proposed and previously approved by the LWG.

We note, for the sake of completeness, that we have used the simplest form of the χ^2 distribution: Although it is possible to extend the (central) χ^2 distribution by allowing non-integral degrees of freedom (n), that form of the distribution is just a special case of the gamma distribution. Only when n is integral is there a more efficient way to generate χ^2 random variates, and so we restrict n to integral values.

10 Conclusion

This paper has described the process and the major decisions leading to [BFKP06], the authors' comprehensive proposal to incorporate a random number generation facility into the C++ standard library. Historical background, rationale, and explanation are provided herein in order to furnish a context in which the Proposal may be evaluated. We respectfully urge that the Proposal be considered on a time scale consistent with its adoption into the Working Paper leading to C++0X.

11 Acknowledgments

We would lilke to acknowledge the Fermi National Accelerator Laboratory's Computing Division, sponsor of our participation in the C++ standards effort, for its past support of our efforts to improve C++ for all our user communities, and for its best efforts to support our continued participation in the face of severe budget constraints.

Bibliography

[Aus05] Matt Austern. (Draft) technical report on standard library extensions. Paper N1836, ISO/IEC SC22/JTC1/WG21, June 24 2005. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf; same as ANSI NCITS/J16 05-0096.

[BD76] Carter Bays and S. D. Durham. Improving a poor random number generator. *ACM Transactions on Mathematics Software*, 2(1):59–64, March 1976.

[BFKP06] Walter E. Brown, Mark Fischler, Jim Kowalkowski, and Marc F. Paterno. Random number generation in C++0x: A comprehensive proposal. Paper N1932, ISO/IEC SC22/JTC1/WG21, March 5 2006. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1932.pdf; same as ANSI NCITS/J16 06-0002.

[Hin05] Howard E. Hinnant. C++ standard library active issues list (revision r40). Paper N1926, ISO/IEC SC22/JTC1/WG21, December 16 2005. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1926.html; same as ANSI NCITS/J16 05-0186.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, USA, third edition, 1998. ISBN 0-201-89684-2. xiii+762 pp.

[Mau03] Jens Maurer. A proposal to add an extensible random number facility to the standard library (revision 2). Paper N1452, ISO/IEC SC22/JTC1/WG21, April 10 2003. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1452.html; same as ANSI NCITS/J16 03-0035.

[Pat04] Marc F. Paterno. On random-number distributions for C++0x. Paper N1588, JTC1-SC22/WG21, February 13 2004. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1588.pdf; same as ANSI NCITS/J16 04-0028.

[PFBK05] Marc Paterno, Mark Fischler, Walter E. Brown, and Jim Kowalkowski. A proposal to add random-number distributions to C++0x. Paper N1914, ISO/IEC SC22/JTC1/WG21, October 21 2005. Online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1914.pdf; same as ANSI NCITS/J16 05-0174.