Recommendations for Resolving the 2007-09-21 Issues re [rand]

Document #: WG21/N2424 = J16/07-0284
Date: 2007-10-02
Revises: None
Project: Programming Language C++
Reference: ISO/IEC IS 14882:2003(E)
Reply to: Walter E. Brown<wb@fnal.gov>
LSC Dept., Computing Division
Fermi National Accelerator Laboratory
Batavia, IL 60510-0500

Contents

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

— DONALD E. KNUTH

1 Introduction

This paper gathers all new issues submitted by Stephan Tolksdorf in his 2007-09-21 posting, and presents recommendations toward their resolution. The text of each issue has been taken verbatim from the posting, and is in some cases lightly reformatted. His item numbers have herein been given a T prefix in order to distinguish his numbering from that used in the LWG Issues List. We have also included commentary (labelled CK) received offline from Charles Karney.

Our contributions are presented in subsections labelled **Recommendations**. Text to be inserted is denoted in red, and text to be deleted is marked like ~~this~~.

2 New Issue T2 = 728: Problem in [rand.eng.mers]/6

Section: [rand.eng.mers] **Submitter:** Stephan Tolksdorf **Date:** 2007-09-21

Discussion: The `mersenne_twister_engine` is required to use a seeding method that is given as an algorithm parameterized over the number of bits `W`. I doubt whether the given generalization of an algorithm that was originally developed only for unsigned 32-bit integers is appropriate for other bit widths. For instance, `W` could be theoretically 16 and `UIntType` a 16-bit integer, in which case the given multiplier would not fit into the `UIntType`. Moreover, T. Nishimura and M. Matsumoto have chosen a different multiplier for their 64 bit Mersenne Twister [reference].

Proposed resolution: I see two possible resolutions:

a) Restrict the parameter `W` of the mersenne_twister_template to values of 32 or 64 and use the multiplier from [the above reference] for the 64-bit case (my preference).

b) Interpret the state array for any `W` as a 32-bit array of appropriate length (and a specified byte order) and always employ the 32-bit algorithm for seeding.

CK comments on 2007-09-25: I agree that the seeding proposed in [paragraph] 6 is overly specific to $w = 32$. However I don't regard this as a major problem because all serious users of the Mersenne twister should use `seed_seq` for seeding, since this gives access to a much larger seed space. `seed_seq` (especially as modified as suggested in Issue 677) does deliver good state for all values of w.

Perhaps a note can be added to the standard recommending the use of `seed_seq` as a seeding mechanism.

A potentially more serious issue is that the tempering described in [rand.eng.mers] does not cover Nishimura and Matsumoto's 64-bit generator. To deal with both the 32-bit and 64-bit versions, [rand.eng.mers] 3a needs to be changed from

a) Let $z_1 = X_i \,\mathsf{xor}\, (X_i \,\mathsf{rshift}\, u)$

to

a) Let $z_1 = X_i \,\mathsf{xor}\, ((X_i \,\mathsf{rshift}\, u) \,\mathsf{bitand}\, d)$

where d is another `UIntType` template parameter equal to `0xffffffff` for the 32-bit version.

Finally, it seems to me that a better generalization of Mersenne twister to arbitrary w would entail changing the last sentence of [rand.eng.mers] para 8 from

changes X_{-n} to 2^{w-1}.

to

changes X_{-n} to 2^r.

[This leaves the 32-bit mt19937 unchanged since $w - 1 = r = 31$. However for the 64-bit version it results in the least significant bit of the 19937-bit state being set instead of somewhat arbitrary 63rd bit. This is a very minor point since `seed_seq::randomize` very rarely produces 19937 consecutive zero bits.]

Recommendations: While the original issue does not seem to us to constitute a defect, CK's comments do contain important improvements in the utility of this engine. We therefore suggest the following resolution to address all aspects of the issue.

As recommended, we first provide additional parameterization to achieve the desired generality. These template parameters are indicated by the following recommended changes in [lib.rand.synopsis] and the same change in the template's synopsis in [lib.rand.eng.mers]:

```
template <class UIntType, size_t w, size_t n, size_t m, size_t r,
          UIntType a, size_t u, UIntType d, size_t s,
          UIntType b, size_t t,
          UIntType c, size_t l, UIntType f>
```

We describe these new parameters in the narrative in [lib.rand.eng.mers]/2:

> ... according to a bit-scrambling matrix defined by values u, d, s, b, t, c, and ℓ.
>
> The sequence X is initialized with the help of an initialization multiplier f.

We publish the new parameters in the template's synopsis in [lib.rand.eng.mers]:

```
static const size_t tempering_u = u;
static const UIntType tempering_d = d;
...
static const size_t tempering_l = l;
static const UIntType initialization_multiplier = f;
```

We use the new parameters in [lib.rand.eng.mers]/3a:

> Let $z_1 = X_i \,\mathsf{xor}\, ((X_i \,\mathsf{rshift}\, u) \,\mathsf{bitand}\, d)$

and in [lib.rand.eng.mers]/6 (replacing the previously-hardcoded constant 1812433253):

$$\left[f \cdot \left(X_{i-1} \,\mathsf{xor}\, (X_{i-1} \,\mathsf{rshift}\, (w-2))\right) + i \bmod n\right] \bmod 2^w \ .$$

In [rand.predef], change the definition of `mt19937`:

```
typedef mersenne_twister_engine<uint_fast32_t,
                                32,624,397,31,0x9908b0df,11,
                                0xffffffff,7,0x9d2c5680,15,0xefc60000,18,
                                1812433253>
        mt19937;
```

Finally, to take advantage of these adjustments, we strongly recommend adding to [rand.predef] an additional `typedef` for a 64-bit `mersenne_twister_engine`, using parameters known from the literature to produce good results. In particular, append after the `typedef` for `mt19937;` in [rand.synopsis]:

```
typedef see below mt19937_64;
```

Also insert after in [rand.predef]/3:

```
typedef mersenne_twister_engine<uint_fast64_t,
                                64,312,156,31,0xb5026f5aa96619e9,29,
                                0x5555555555555555,17,
                                0x71d67fffeda60000,37,
                                0xfff7eee000000000,43,
                                6364136223846793005>
        mt19937_64;
```

Required behavior: The 100000^{th} consecutive invocation of a default_constructed object of type mt19937_64 shall produce the value 14002232017267485025.

3 New Issue T3 = 729: Problem in [rand.req.eng]/3

Section: [rand.req.eng] **Submitter:** Stephan Tolksdorf **Date:** 2007-09-21

Discussion: The 3rd table row in [rand.req.eng]/3 requires random number engines to accept any arithmetic type as a seed, which is then casted to the engine's `result_type` and subsequently used for seeding the state of the engine. The requirement stated as "Creates an engine with initial state determined by `static_cast<X::result_type>(s)`" forces random number engines to either use a seeding method that completely depends on the `result_type` (see the discussion of seeding for the `mersenne_twister_engine` in point T2 above) or at least to throw away "bits of randomness" in the seed value if the `result_type` is smaller than the seed type. This seems to be inappropriate for many modern random number generators, in particular F2-linear or cryptographic ones, which operate on an internal bit array that in principle is independent of the type of numbers returned.

Proposed resolution: I propose to change the wording to a version similar to "Creates an engine with initial state determined by `static_cast<UintType>(s)`, where `UintType` is an implementation specific unsigned integer type."

Additionally, the definition of `s` in [rand.req.eng]/1 c) could be restricted to unsigned integer types.

Similarly, the type of the seed in [rand.req.adapt]/3 e) could be left unspecified.

CK comments on 2007-09-25: I do not regard this as a serious issue as single-word seeding is only suitable for casual use and `seed_seq` provides access to a much bigger pool of seeds. For casual use having a limited but fully specified definition is preferable to having a potentially better but implementation-specific one.

For most applications, users should be steered away from generators with only 1 or 2 words of state, and `seed_seq` should be the recommended procedure for seeding generators with multi-word states.

ST comments on 2007-09-30: The proposed resolution would not violate the design goal of portability and reproducibility, if the relaxed (general) engine requirements were accompanied with appropriate additional specifications for the specific engines described in the standard.

Recommendations: We recommend these related issues be closed as NAD.

The main issue is not a problem, because the `seed_seq` type was devised precisely to handle such a situation: Given a large integer, supply it to a `seed_seq` constructor, and then furnish the newly-constructed `seed_seq` to the engine.

Further, the first and third parts of the proposed resolution would violate an important design goal for the random number portion of the standard library: portability and reproducibility across implementations.

The second part of the resolution is in fact feasible, but we believe there is no good reason to restrict the type as suggested.

4 New Issue T4 = 730: Comment on [rand.req.adapt]/3 e)

Section: [rand.req.adapt] **Submitter:** Stephan Tolksdorf **Date:** 2007-09-21

Discussion: If an engine adaptor is invoked with an argument of type `seed_seq`, then all base engines are specified to be seeded with this `seed_seq`. As `seed_seq`'s randomization method is qualified as constant, this procedure will effectively initialize all base engines with the same seed (though the resulting state might still differ to a certain degree if the engines are of different types). It is not clear whether this mode of operation is in general appropriate, hence — as far as the stated requirements are of general nature and not just specific to the engine adaptors provided by the library — it might be better to leave the behaviour unspecified, since the current definition of `seed_seq` does not allow for a generally satisfying specification.

Proposed resolution: [As above]

CK comments on 2007-09-25: This limitation of the seeding mechanism by `seed_seq` (which applies equally to the case when seeding via an integer) is clear. However, the seeding mechanism in [rand.req.adapt] para 4 allows independent seeding of the individual generators, so I don't regard this limitation as a strong argument for changing the constness `seed_seq::randomize`.

Recommendations: Of the adaptors currently in the Working Paper, only one adapts multiple engines. Therefore, for most of the standard library, the above issue does not arise. The sole remaining adaptor is `xor_combine_engine`, an engine already known to have terrible properties when used to adapt two engines of like type. Therefore, the above issue is not new in this case.

In addition, the proposed resolution is unacceptable, as it would violate our design goal of portability and reproducibility across implementations.

Also, as CK points out, there already exists another constructor that can initialize all the base engines independently.

Finally, we have no experience with any multi-engine adaptors that behave well when using engines of like type. We are therefore uncertain as to the best way to cope.

For all the above reasons, we recommend this issue be closed as NAD.

5 New Issue T6 = 731: proposal for a customizable `seed_seq`

Section: [rand.util.seedseq] **Submitter:** Stephan Tolksdorf **Date:** 2007-09-21

Discussion: The proper way to seed random number engines seems to be the most frequently discussed issue of the [rand] proposal. While the new `seed_seq` approach is already rather general and probably sufficient for most situations, it is unlikely to be optimal in every case (one problem was pointed out in point T5 above). In some situations it might, for instance, be better to seed the state with a cryptographic generator.

In my opinion this is a pretty strong argument for extending the standard with a simple facility to customize the seeding procedure. This could, for example, be done with the following minimal changes:

Proposed resolution:

a) Turn the interface specification of [rand.util.seedseq]/2 into a "SeedSeq" requirement, where the exact behaviour of the constructors and the randomize method are left unspecified and where the

const qualification for randomize is removed. Classes implementing this interface are additionally required to specialize the traits class in c).

b) Provide the class `seed_seq` as a default implementation of the SeedSeq interface.

c) Supplement the `seed_seq` with a traits class

```
template <typename T>
struct is_seed_seq { static const bool value = false; }
```

and the specialization

```
template <>
struct is_seed_seq<seed_seq> { static const bool value = true; }
```

which users can supplement with further specializations.

d) Change [rand.req.eng]/1 d) to "q is an lvalue of a type that fulfils the SeedSeq requirements", and modify the constructors and seed methods in [rand.eng] appropriately (the actual implementation could be done using the SFINAE technique).

Recommendations: We agree with the underlying intent of this issue, but believe it does not constitute a defect.

However, we further believe that Concepts provide a more appropriate mechanism to obtain the desired effect. (For example, it would obviate the need for and the utility of the traits class proposed above.) We therefore recommend that this issue be revisited in that context at a more appropriate juncture.

6 New Issue T7 = 732: Defect in [rand.dist.samp.genpdf]

Section: [rand.dist.samp.genpdf] **Submitter:** Stephan Tolksdorf **Date:** 2007-09-21

Discussion: [rand.dist.samp.genpdf] describes the interface for a distribution template that is meant to simulate random numbers from any general distribution given only the density and the support of the distribution. I'm not aware of any general purpose algorithm that would be capable of correctly and efficiently implementing the described functionality. From what I know, this is essentially an unsolved research problem. Existing algorithms either require more knowledge about the distribution and the problem domain or work only under very limited circumstances. Even the state of the art special purpose library UNU.RAN does not solve the problem in full generality, and in any case, testing and customer support for such a library feature would be a nightmare.

Proposed resolution: For these reasons, I propose to delete section [rand.dist.samp.genpdf].

Recommendations: We do not agree that there is any defect.

While we agree that not all functions can be well-handled, we believe that the set of functions that can be handled with sufficient accuracy is large enough to make this distribution valuable.

For the record, we have two working implementations of this distribution, both with acceptable fidelity on non-pathological functions. We believe that the degree of fidelity and the degree of ability to cope with pathology are QOI issues.

7 New Issue T8 = 733: Comment on [rand.req.dist]/9

Section: [rand.req.dist] **Submitter:** Stephan Tolksdorf **Date:** 2007-09-21

Discussion: The requirement "P shall have a declaration of the form `typedef X distribution_type`" effectively makes the use of inheritance for implementing distributions very inconvenient, because the child of a distribution class in general will not satisfy this requirement. In my opinion the benefits of having a typedef in the parameter class pointing back to the distribution class are not worth the hassle this requirement causes. [In my code base I never made use of the nested typedef but on several occasions could have profited from being able to use simple inheritance for the implementation of a distribution class.]

Proposed resolution: I propose to drop this requirement.

Recommendations: Our implementation makes no use of these `typedef`s, either, but that's not why they are present. The motivation for the `typedef`s was for use in generic code that would be handed an object of arbitrary parameter type. Given such an object, this seemed the most straightforward way for the generic algorithm to determine the type of the distribution it can construct from that parameter instance.

We believe this issue does not rise to the level of a defect.

8 New Issue T9 = 734: Unnecessary restriction in [rand.dist.norm.chisq, f, t]

Section: [rand.dist.norm.chisq, f, t] **Submitter:** Stephan Tolksdorf **Date:** 2007-09-21

Discussion: `chi_squared_distribution`, `fisher_f_distribution` and `student_t_distribution` have parameters for the "degrees of freedom" `n` and `m` that are specified as integers. For the following two reasons this is an unnecessary restriction: First, in many applications such as Bayesian inference or Monte Carlo simulations it is more convenient to treat the respective parameters as continuous variables. Second, the standard non-naive algorithms (i.e. $\mathcal{O}(1)$ algorithms) for simulating from these distributions work with floating-point parameters anyway (all three distributions could be easily implemented using the Gamma distribution, for instance).

Similar arguments could in principle be made for the parameters `t` and `k` of the `discrete binomial_distribution` and `negative_binomial_distribution`, though in both cases continuous parameters are less frequently used in practice and in case of the `binomial_distribution` the implementation would be significantly complicated by a non-discrete parameter (in most implementations one would need an approximation of the log-gamma function instead of just the log-factorial function).

Proposed resolution: For these reasons, I propose to change the type of the respective parameters to `double`.

Recommendations: We agree that these distributions could easily be extended as suggested, and would be amenable to doing so if LWG so recommends. We note that testing the extended versions from basic principles would be significantly harder than it is for the current integral versions.

9 New Issue T10 = 735: Unfortunate naming

Section: [rand.dist.bern.bin] and [rand.dist.bern.negbin] **Submitter:** Stephan Tolksdorf **Date:** 2007-09-21

Discussion: In my opinion the choice of name for the `t` parameter of the binomial_distribution is very unfortunate. In virtually every internet reference, book and software implementation this parameter is called `n` instead, see for example Wikipedia, Mathworld, Evans et al. (1993) Statistical Distributions, 2nd E., Wiley, p. 38, the R statistical computing language, p. 926, Mathematica and Matlab.

Similarly, the choice of `k` for the parameter of the negative binomial distributions is rather unusual. The most common choice for the negative binomial distribution seems to be `r` instead.

Choosing unusual names for the parameters causes confusion among users and makes the interface unnecessarily inconvenient to use.

Proposed resolution: For these reasons, I propose to change the name of the respective parameters to `n` and `r`.

Recommendations: Naming is important, and so we would be willing to support the proposed resolution, especially for the `binomial_distribution`. There seems much less consensus in the literature for the `negative_binomial_distribution`

In any event, we believe that the issue is editorial. We therefore recommend it be forwarded to the Project Editor for resolution as he sees fit.

10 New Issue T11 = 736: Comment on [rand.dist.samp.discrete]

Section: [rand.dist.samp.discrete] **Submitter:** Stephan Tolksdorf **Date:** 2007-09-21

Discussion: a) The specification for `discrete_distribution` requires the member `probabilities()` to return a vector of *standardized* probabilities, which forces the implementation every time to divide each probability by the sum of all probabilities, as the sum will in practice almost never be exactly 1.0. This is unnecessarily inefficient as the implementation would otherwise not need to compute the standardized probabilities at all and could instead work with the non-standardized probabilities and the sum. If there was no standardization the user would just get back the probabilities that were previously supplied to the distribution object, which to me seems to be the more obvious solution.

b) The behaviour of `discrete_distribution` is not specified in case the number of given probabilities is larger than the maximum number representable by the IntType.

Proposed resolution: I propose to change the specification such that the non-standardized probabilities need to be returned and that an additional requirement is included for the number of probabilities to be smaller than the maximum of IntType.

Recommendations: We agree with the observation and the proposed resolution to part b). We recommend the wording $n > 0$ be replaced with $0 < n \leq$ `numeric_limits<IntType>::max()` $+ 1$.

However, we disagree with part a), as it would intefere with the definition of parameters' equality. Further, the changed requirement would lead to a significant increase in the amount of state of the distribution object.

11 New Issue T12 = 737: Comment on [rand.dist.samp.pconst]

Section: [rand.dist.samp.pconst] **Submitter:** Stephan Tolksdorf **Date:** 2007-09-21

Discussion: a) The discussion in point T11 above regarding `probabilities()` similarly applies to the method `densities()` of `piecewise_constant_distribution`.

b) The design of the constructor

```
template <class InputIteratorB, class InputIteratorW>
piecewise_constant_distribution( InputIteratorB firstB, InputIteratorB lastB,
                                 InputIteratorW firstW);
```

is unnecessarily unsafe, as there is no separate end-iterator given for the weights. I can't see any performance or convenience reasons that would justify the risks inherent in such a function interface, in particular the risk that input error might go unnoticed.

Proposed resolution: I propose to add an `InputIteratorW lastW` argument to the interface.

Recommendations: Regarding issue a), we take the same position as we did in the referenced issue, and disagree for the same reasons.

Regarding issue b), we observe that the current design is consistent with comparable usage elsewhere in the standard library. We therefore recommend against the proposed resolution.

12 New Issue T13 = 738: Editorial issue in [rand.adapt.disc]/3

Section: [rand.adapt.disc] **Submitter:** Stephan Tolksdorf **Date:** 2007-09-21

Discussion: Since the template parameter `p` and `r` are of type `size_t`, the member `n` in the class exposition should have type `size_t`, too.

Proposed resolution: [As above]

Recommendations: We agree that the value in question is always non-negative, that type `size_t` would be a reasonable (and arguably better) choice, and that the issue is editorial. We recommend it be forwarded to the Project Editor for resolution as he sees fit.

13 New Issue T14 = 739: Defect in [rand.util.canonical]/3

Section: [rand.util.canonical] **Submitter:** Stephan Tolksdorf **Date:** 2007-09-21

Discussion: The complexity of `generate_canonical` is specified to be "exactly k=max(1, ceil(b/log2 R)) invocations of g". This terms involves a logarithm that is not rounded and hence can not (in general) be computed at compile time. As this function template is performance critical, I propose to replace ceil(b/log2 R) with ceil(b/floor(log2 R)).

Proposed resolution: [As above]

ST comments on 2007-09-30: Although the proposed resolution would occasionally yield results for k which are different from the original calculation, these results would always be at least as

high as the original values and hence could be viewed as conservative estimates. However, I agree that it suffices to calculate k at most once per instantiation and that the issue thus is not a defect.

Recommendations: For the record, the formula under discussion originated from the following definition of k: Let k be the smallest integer such that any k-digit base R number, when expressed as a binary number, will consist of at least b bits. Formulated mathematically, the desired relation is $R^k - 1 \geq 2^b - 1$, which is equivalent to $R^k \geq 2^b$. Taking logarithms, we have $\log_R R^k \geq \log_R 2^b$, which simplifies to $k \geq \log_R 2^b$. Converting the base of the logarithm produces $k \geq \log_2 2^b / \log_2 R$, simplifying to $k \geq b / \log_2 R$. Expressed as an equality, we obtain $k = \lceil b / \log_2 R \rceil$, as specified in the Working Paper.

We next observe that there is no requirement that the value of k be calculated at compile-time (or ever, for that matter). Further, the proposed resolution would occasionally yield different results from the original calculation, and those different results would be incorrect. For example, when $R = 10$ and $b = 53$, the WP's formula gives 16, while the proposed formula yields 18.

We therefore recommend that this issue be closed as NAD.

14 Summary and conclusion

This paper has recommended resolutions to issues, submitted by Stephan Tolksdorf, related to random number generation in the standard library. It is our firm hope that these recommendations receive the attention of the Library Working Group on a time scale commensurate with final adoption into C++0X.

15 Acknowledgments

I gratefully acknowledge the able advice and assistance of my colleagues Mark Fischler and Marc Paterno, and also thank the Fermi National Accelerator Laboratory's Computing Division, sponsor of our participation in the C++ standards effort, for its past and continuing support of our efforts to improve C++ for all our user communities.