

# Recommendations for Resolving Issues re [rand]

*Document #:* WG21/N2391 = J16/07-0251  
*Date:* 2007-09-07  
*Revises:* None  
*Project:* Programming Language C++  
*Reference:* ISO/IEC IS 14882:2003(E)  
*Reply to:* Walter E. Brown<[wb@fnal.gov](mailto:wb@fnal.gov)>  
LSC Dept., Computing Division  
*Fermi National Accelerator Laboratory*  
Batavia, IL 60510-0500

---

## Contents

<b>1 Introduction</b>	<b>1</b>
<b>2 Issue 552: <code>random_shuffle</code> and its generator</b>	<b>2</b>
<b>3 Issue 699: N2111 changes <code>min/max</code></b>	<b>2</b>
<b>4 Issue 654: Missing IO roundtrip for random number engines</b>	<b>3</b>
<b>5 Issue 678: Changes for <code>[rand.req.eng]</code></b>	<b>5</b>
<b>6 Issue 548: May <code>random_device</code> block?</b>	<b>6</b>
<b>7 Issue 608: Unclear <code>seed_seq</code> construction details</b>	<b>6</b>
<b>8 Issue 607: Concern about short seed vectors</b>	<b>7</b>
<b>9 Issue 677: Weaknesses in <code>seed_seq::randomize</code></b>	<b>8</b>
<b>10 Issue 712: <code>seed_seq::size</code> no longer useful</b>	<b>11</b>
<b>11 Issue 655: Signature of <code>generate_canonical</code> not useful</b>	<b>11</b>
<b>12 Summary and conclusion</b>	<b>12</b>
<b>13 Acknowledgments</b>	<b>12</b>

---

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

— DONALD E. KNUTH

## 1 Introduction

Recognizing that the Library Working Group is laboring under considerable time pressure to complete its work on C++0X, this paper gathers all known issues related to the random number portion of the standard library, and presents recommendations toward their resolution. The text of each issue is taken verbatim from [N2403](#), and is in some cases lightly reformatted. 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 Issue 552: random\_shuffle and its generator

**Section:** 25.2.12 [alg.random.shuffle] **Submitter:** Martin Sebor **Date:** 2006-01-25

**Discussion:** ... is specified to shuffle its range by calling `swap` but not how (or even that) it's supposed to use the `RandomNumberGenerator` argument passed to it.

Shouldn't we require that the generator object actually be used by the algorithm to obtain a series of random numbers and specify how many times its `operator()` should be invoked by the algorithm?

**Proposed resolution:** [none]

**Recommendations:** We agree that the `random_shuffle` algorithm seems a bit underspecified with respect to this issue. However, the specification for the third form of the algorithm should be phrased in terms of the appropriate distribution, rather than in terms of the supplied generator: this permits the possibility of invoking the generator fewer times by a sufficiently clever algorithm. We propose the following changes to the specification, also simplifying a bit of the verbiage:

*4 Remarks:* 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.

The second form of the function takes a random number generating function object *rand* such that if *n* is ~~an argument for *rand*, with a positive value, that has~~ of type `iterator_traits<RandomAccessIterator>::difference_type`, then *rand*(*n*) returns a randomly chosen value, ~~which lies in the interval [0,n), and which has~~ of a type that is convertible to `iterator_traits<RandomAccessIterator>::difference_type`. **This form of the function makes  $\max(0, (last - first) - 1)$  calls to *rand*.**

The third form of the function takes an object *g* meeting the requirements of uniform random number generator (26.4.1.2). **This form of the function makes  $\max(0, n - 1)$  calls to *d*(*g*), where *n* is *last - first*, and *d* is an object of type `uniform_int_distribution<iterator_traits<RandomAccessIterator>::difference_type>` (**[rand.dist.uni.int]) that was constructed with arguments (0, n).****

## 3 Issue 699: N2111 changes min/max

**Section:** 26.4 [rand] **Submitter:** P.J. Plauger **Date:** 2007-07-01

**Discussion:** N2111 [2006-10-19] changes `min/max` in several places in `random` from member functions to static data members. I believe this introduces a needless backward compatibility problem between C++0X and TR1. I'd like us to find new names for the static data members, or perhaps change `min/max` to `constexprs` in C++0X.

**Proposed resolution:** [none]

**Recommendations:** While we understand and are sympathetic to this argument, we believe this issue does not constitute a defect in the Working Paper or in the Standard, and therefore recommend that this issue be closed NAD.

However, we further recommend (and intend to undertake) a separate review of the random number portion of the library, with an eye toward the application of new core language features

such as concepts, `constexpr`, etc. It seems likely that the outcome of such a review will recommend adjustments in `min/max` and/or other members in the direction this issue sought.

#### 4 Issue 654: Missing IO roundtrip for random number engines

**Section:** 26.4.1.3 [rand.req.eng] **Submitter:** Daniel Krügler **Date:** 2007-03-08

**Discussion:** Table 98 and para 5 in 26.4.1.3 [rand.req.eng] specify the IO insertion and extraction semantic of random number engines. It can be shown, v.i., that the specification of the extractor cannot guarantee to fulfill the requirement from para 5:

If a textual representation written via `os << x` was subsequently read via `is >> v`, then `x == v` provided that there have been no intervening invocations of `x` or of `v`.

The problem is, that the extraction process described in table 98 misses to specify that it will initially set the `if.fmtflags` to `ios_base::dec`, see table 104:

`dec`: converts integer input or generates integer output in decimal base

**Proof:** The following small program demonstrates the violation of requirements (exception safety not fulfilled):

```
#include <cassert>
#include <ostream>
#include <iostream>
#include <iomanip>
#include <sstream>

class RanNumEngine {
    int state;
public:
    RanNumEngine() : state(42) {}

    bool operator == (RanNumEngine other) const {
        return state == other.state;
    }

    template <typename Ch, typename Tr>
    friend std::basic_ostream<Ch, Tr>&
    operator << (std::basic_ostream<Ch, Tr>& os, RanNumEngine engine) {
        Ch old = os.fill(os.widen(' ')); // Sets space character
        std::ios_base::fmtflags f = os.flags();
        os << std::dec << std::left
            << engine.state; // Adds ios_base::dec|ios_base::left
        os.fill(old); // Undo
        os.flags(f);
        return os;
    }

    template <typename Ch, typename Tr>
    friend std::basic_istream<Ch, Tr>&
    operator >> (std::basic_istream<Ch, Tr>& is, RanNumEngine& engine) {
        // Uncomment only for the fix.
```

```

        //std::ios_base::fmtflags f = is.flags();
        //is >> std::dec;
        is >> engine.state;
        //is.flags(f);
        return is;
    }
};

int main() {
    std::stringstream s;
    s << std::setfill('#'); // No problem
    s << std::oct; // Yikes!
    // Here starts para 5 requirements:
    RanNumEngine x;
    s << x;
    RanNumEngine v;
    s >> v;
    assert(x == v); // Fails: 42 == 34
}

```

A second, minor issue seems to be, that the insertion description from table 98 unnecessarily requires the addition of `ios_base::fixed` (which only influences floating-point numbers). Its not entirely clear to me whether the proposed standard does require that the state of random number engines is stored in integral types or not, but I have the impression that this is the indent, see e.g. p. 3

The specification of each random number engine defines the size of its state in multiples of the size of its `result_type`.

If other types than integrals are supported, then I wonder why no requirements are specified for the precision of the stream.

#### Proposed resolution:

1) In table 98 from 26.4.1.3 [rand.req.eng] in column “pre/post-condition”, row expression “`is >> x`” change

Sets `v`'s state as determined by reading its textual representation **with `is.fmtflags` set to `ios_base::dec`** from `is`.

2) In table 98 from 26.4.1.3 [rand.req.eng] in column “pre/post-condition”, row expression “`os << x`” change

With `os.fmtflags` set to `ios_base::dec`~~`ios_base::fixed`~~`ios_base::left` and `[..]`

**Recommendations:** This relatively minor 2-part issue proposes to tighten the requirements on the manipulators to be used while inserting/extracting a random number engine. We agree with the analyses, and with the proposed resolutions. However, for improved consistency in wording, we recommend the following slightly different formulation:

1) In Table 98, row `is >> v`:

**S**With `is.fmtflags` set to `ios_base::dec`, sets `v`'s state as determined by reading its textual representation from `is`.

2) As proposed above.

## 5 Issue 678: Changes for [rand.req.eng]

**Section:** 26.4.1.3 [rand.req.eng] **Submitter:** Charles Karney **Date:** 2007-05-15

**Discussion:** This change follows naturally from the proposed change to `seed_seq::randomize` in [issue] 677.

In table 104 the description of  $X(q)$  contains a special treatment of the case `q.size() == 0`. This is undesirable for 4 reasons:

1. It replicates the functionality provided by  $X()$ .
2. It leads to the possibility of a collision in the state provided by some other  $X(q)$  with `q.size() > 0`.
3. It is inconsistent with the description of the  $X(q)$  in paragraphs 26.4.3.1 [rand.eng.lcong] p5, 26.4.3.2 [rand.eng.mers] p8, and 26.4.3.3 [rand.eng.sub] p10 where there is no special treatment of `q.size() == 0`.
4. The proposed replacement for `seed_seq::randomize` given above allows for the case `q.size() == 0`.

**Proposed resolution:** I recommend removing the special-case treatment of `q.size() == 0`. Here is the replacement line for table 104 of section 26.4.1.3 [rand.req.eng]:

$X(q)$	<del>With <math>n = q.size()</math>, creates an engine <math>u</math> with initial state determined as follows: If <math>n</math> is 0, <math>u == X()</math>; otherwise, the</del> <b>Create an engine <math>u</math> with an initial state which depends on a sequence produced by one call to <code>q.randomize</code>.</b>	$\mathcal{O}(\max(n, q.size()),$ size of state)
--------	---	--

**Recommendations:** As pointed out above, this issue is closely related to issue 677 ( $q.v.$ ). If issue 677 is closed NAD, then we recommend the same for this issue. However, assuming our recommendations for issue 677 are adopted, we recommend addressing this issue as shown below, making a minor adjustment to the effects and updating the time complexity.

We endorse the proposed resolution. but for a reason not enumerated above. The underlying philosophical issue is simply stated: Do we want engine initialization via a `seed_seq` constructed from an empty vector to be identical with an engine's default-initialization? We previously thought this was desirable, but after reflecting on this issue we are now of the opinion that constructing an engine from a `seed_seq` should not depend on how the `seed_seq` was initialized.

Accordingly, we recommend the following changes for table 104 of section 26.4.1.3 [rand.req.eng]:

$X(q)$	<del>With <math>n = q.size()</math>, creates</del> <b>Create</b> an engine $u$ with <del>initial state determined as follows: If <math>n</math> is 0, <math>u == X()</math>; otherwise, the</del> <b>an</b> initial state <b>that</b> depends on a sequence produced by one call to <code>q.randomize</code> .	$\mathcal{O}(\max(n, size$ of state))
--------	--	--

## 6 Issue 548: May random\_device block?

**Section:** 26.4.6 [rand.device], TR1 5.1.6 [tr.rand.device] **Submitter:** Matt Austern **Date:** 2006-01-10

**Discussion:** Class `random_device` “produces non-deterministic random numbers”, using some external source of entropy. In most real-world systems, the amount of available entropy is limited. Suppose that entropy has been exhausted. What is an implementation permitted to do? In particular, is it permitted to block indefinitely until more random bits are available, or is the implementation required to detect failure immediately? This is not an academic question. On Linux a straightforward implementation would read from `/dev/random`, and “When the entropy pool is empty, reads to `/dev/random` will block until additional environmental noise is gathered.” Programmers need to know whether `random_device` is permitted to (or possibly even required to?) behave the same way.

[ Berlin: Walter: N1932 considers this NAD. Does the standard specify whether `std::cin` may block? ]

**Proposed resolution:** [none]

**Recommendations:** We continue to view the situation with `std::cin` as analogous, as both *de facto* behave as input sources. Unless/until the Working Paper intends to specify whether `std::cin` may block, we believe this issue should be closed NAD.

## 7 Issue 608: Unclear seed\_seq construction details

**Section:** 26.4.7.1 [rand.util.seedseq] **Submitter:** Charles Karney **Date:** 2006-10-26

**Discussion:** In 26.4.7.1 [rand.util.seedseq] /6, the order of packing the inputs into  $b$  and the treatment of signed quantities is unclear. Better to spell it out.

**Proposed resolution:**

$$b = \text{sum}(\text{unsigned}(\text{begin}[i]) \cdot 2^{(w \cdot i)}, 0 \leq i < \text{end} - \text{begin})$$

where  $w$  is the bit-width of the `InputIterator`.

**Recommendations:** We believe this issue is not about clarity, but about defining certain unspecified behavior. We agree that the behavior ought to be specified, and via the general approach suggested above. We recommend the following reformulation, as it provides a tighter and more consistent integration with the existing wording:

6 *Effects:* Constructs a `seed_seq` object by rearranging the bits of the supplied sequence `[begin, end)` of  $w$ -bit quantities into 32-bit units, as if by first concatenating all the  $n$  bits that make up the supplied sequence to initialize a single (possibly very large) unsigned binary number,  $b = \sum_{i=0}^{n/w} \text{begin}[i] \cdot 2^{w \cdot i}$  (in which the bits of each `begin[i]` are treated as denoting an unsigned quantity), and then carrying out the following algorithm:

```
for( v.clear(); n > 0; n -= 32 )
    v.push_back( b mod 232, b /= 232;
```

## 8 Issue 607: Concern about short seed vectors

**Section:** 26.4.7.1 [rand.util.seedseq] **Submitter:** Charles Karney **Date:** 2006-10-26

**Discussion:** Short seed vectors of 32-bit quantities all result in different states. However this is not true of seed vectors of 16-bit (or smaller) quantities. For example these two seeds

```
unsigned short seed = {1, 2, 3};
unsigned short seed = {1, 2, 3, 0};
```

both pack to

```
unsigned seed = {0x20001, 0x3};
```

yielding the same state.

**Proposed resolution:** In 26.4.7.1[rand.util.seedseq]/8a, replace

Set `begin[0]` to  $5489 + sN$ , where  $N$  is the bit length of the sequence used to construct the `seed_seq` in 26.4.7.1/6 [rand.util.seedseq]. (This quantity is called  $n$  in 26.4.7.1/6 [rand.util.seedseq], but  $n$  has a different meaning in 26.4.7.1/8 [rand.util.seedseq]. We have  $32^{(s-1)} < N \leq 32^s$ .) Now

```
unsigned short seed = {1, 2, 3, 0};
unsigned seed = {0x20001, 0x3};
```

are equivalent ( $N = 64$ ), but

```
unsigned short seed = {1, 2, 3};
```

gives a distinct state ( $N = 48$ ).

**Recommendations:** We first observe that this issue would become moot if issue 677 is adopted: a part of 677's proposed resolution would completely replace 26.4.7.1/8a, the paragraph most central to the present issue's resolution. However, in the event that we do not adopt issue 677, we present our recommendations below.

We concur with the above analysis, and agree with the direction of the proposed resolution. However, we believe the proposed wording is somewhat argumentative, and therefore instead recommend as follows:

In 26.4.7.1/1, insert as marked:

```
1 An object of type seed_seq consumes a sequence of integer-valued data, w bits each,
and produces a fixed number of unsigned integer values,  $0 \leq i < 2^{32}$ , based on the
consumed data. [ Note: ...—end note ]
```

In the class synopsis following 26.4.7.1/2, insert a new private data member:

```
size_t sz; // exposition only
vector<result_type> v; // exposition only
```

In 26.4.7.1/3, insert as marked:

3 *Effects*: Constructs a `seed_seq` object as if by default-constructing its member `s` `sz` and `v`.

In 26.4.7.1/6, change as marked:

6 *Effects*: Constructs a `seed_seq` object by rearranging the bits of the supplied sequence `[begin, end)` into 32-bit units, as if by first concatenating all the `sz = n` bits that make up the supplied sequence to initialize a single (possibly very large) unsigned binary number, `b`, and then carrying out the following algorithm:

```
for( v.clear(); n > 0; n -= 32 )
    v.push_back( b mod 232, b /= 232;
```

In 26.4.7.1/8a, change as marked:

7 a) Set `begin[0]` to `5489 + s sz`. ...

## 9 Issue 677: Weaknesses in `seed_seq::randomize`

**Section:** 26.4.7.1 [rand.util.seedseq] **Submitter:** Charles Karney **Date:** 2007-05-15

**Discussion:** `seed_seq::randomize` provides a mechanism for initializing random number engines which ideally would yield "distant" states when given "close" seeds. The algorithm for `seed_seq::randomize` given in the current Working Draft for C++, N2284 (2007-05-08), has 3 weaknesses

1. Collisions in state. Because of the way the state is initialized, seeds of different lengths may result in the same state. The current version of `seed_seq` has the following properties:

- For a given  $s \leq n$ , each of the  $2^{32s}$  seed vectors results in a distinct state.

The proposed algorithm (below) has the considerably stronger properties:

- All of the  $(2^{32n} - 1)/(2^{32} - 1)$  seed vectors of lengths  $s < n$  result in distinct states.
- All of the  $2^{32n}$  seed vectors of lengths  $s = n$  result in distinct states.

2. Poor mixing of `v`'s entropy into the state. Consider `v.size() == n` and hold `v[n/2]` thru `v[n-1]` fixed while varying `v[0]` thru `v[n/2-1]`, a total of  $2^{16n}$  possibilities. Because of the simple recursion used in `seed_seq`, `begin[n/2]` thru `begin[n-1]` can take on only  $2^{64}$  possible states.

The proposed algorithm uses a more complex recursion which results in much better mixing.

3. `seed_seq::randomize` is undefined for `v.size() == 0`. The proposed algorithm remedies this.

The current algorithm for `seed_seq::randomize` is adapted by me from the initialization procedure for the Mersenne Twister by Makoto Matsumoto and Takuji Nishimura. The weakness (2) given above was communicated to me by Matsumoto last year.

The proposed replacement for `seed_seq::randomize` is due to Mutsuo Saito, a student of Matsumoto, and is given in the implementation of the SIMD-oriented Fast Mersenne Twister random number generator SFMT.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html>

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/SFMT-src-1.2.tar.gz>



See Mutsuo Saito, An Application of Finite Field: Design and Implementation of 128-bit Instruction-Based Fast Pseudorandom Number Generator, Master's Thesis, Dept. of Math., Hiroshima University (Feb. 2007)

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/M062821.pdf>

One change has been made here, namely to treat the case of small  $n$  (setting  $t = (n-1)/2$  for  $n < 7$ ).

Since `seed_seq` was introduced relatively recently there is little cost in making this incompatible improvement to it.

**Proposed resolution:** The following is the proposed replacement of paragraph 8 of section 26.4.7.1 [rand.util.seedseq]:

8 *Effects:* With  $s = v.size()$  and  $n = end - begin$ , fills the supplied range  $[begin, end)$  according to the following algorithm in which each operation is to be carried out modulo  $2^{32}$ , each indexing operator applied to  $begin$  is to be taken modulo  $n$ , ~~each indexing operator applied to  $v$  is to be taken modulo  $s$ , and  $T(x)$  is defined as  $x \text{ xor } (x \text{ rshift } 3027)$ :~~

- a) ~~Set  $begin[0]$  to  $5489 + s$ . Then, iteratively for  $k = 1, \dots, n - 1$ , set  $begin[k]$  to~~

~~$1812433253 * T(begin[k-1]) + k$ .~~

- b) ~~With  $m$  as the larger of  $s$  and  $n$ , transform each element of the range (possibly more than once): iteratively for  $k = 0, \dots, m - 1$ , set  $begin[k]$  to~~

~~$(begin[k] \text{ xor } (1664525 * T(begin[k-1]))) + v[k] + (k \text{ mod } s)$ .~~

- c) ~~Transform each element of the range one last time, beginning where the previous step ended: iteratively for  $k = m \text{ mod } n, \dots, n - 1, 0, \dots, (m - 1) \text{ mod } n$ , set  $begin[k]$  to~~

~~$(begin[k] \text{ xor } (1566083941 * T(begin[k-1]))) - k$ .~~

`fill(begin, end, 0x8b8b8b8b);`

`if (n >= 623)`

`t = 11;`

`else if (n >= 68)`

`t = 7;`

`else if (n >= 39)`

`t = 5;`

`else if (n >= 7)`

`t = 3;`

`else`

`t = (n-1)/2;`

`p = (n-t)/2;`

`q = p+t;`

`m = max(s+1, n);`

`for (k = 0; k < m; k += 1) {`

`r = 1664525 * T(begin[k] ^ begin[k+p] ^ begin[k-1]);`

`begin[k+p] += r;`

`r += k % n;`

```

    if (k == 0)
        r += s;
    else if (k <= s)
        r += v[k-1];
    begin[k+q] += r;
    begin[k] = r;
}

for (k = m; k < m + n; k += 1) {
    r = 1566083941 * T(begin[k] + begin[k+p] + begin[k-1]);
    begin[k+p] ^= r;
    r -= k % n;
    begin[k+q] ^= r;
    begin[k] = r;
}

```

**Recommendations:** It seems clear from the citations provided in the above discussion that the state of the art has advanced since we formulated our proposal N2111 (2006-10-19). We are fortunate to be able to incorporate a provably better algorithm into C++0X, and generally support the revision proposed above.

However, we believe the level of detail in the proposed resolution is overly specific as to the coding required of an implementation. We therefore recommend the following reformulation of the proposed algorithm, preserving the style of the algorithm it replaces:

8 *Effects:* With  $s = v.size()$  and  $n = end - begin$ , fills the supplied range  $[begin, end)$  according to the following algorithm in which each operation is to be carried out modulo  $2^{32}$ , each indexing operator applied to  $begin$  is to be taken modulo  $n$ , ~~each indexing operator applied to  $v$  is to be taken modulo  $s$~~ , and  $T(x)$  is defined as  $x \text{ xor } (x \text{ rshift } 3027)$ :

a) ~~Set  $begin[0]$  to  $5489 + s$ . Then, iteratively for  $k=1, \dots, n-1$ , set  $begin[k]$  to~~

$$\text{1812433253} * T(\text{begin}[k-1]) + k.$$

**By way of initialization, set each element of the range to the value  $0x8b8b8b8b$ . Additionally, for use in subsequent steps, let  $p = (n - t)/2$  and let  $q = p + t$ , where**

$$t = (n \geq 623) ? 11 : (n \geq 68) ? 7 : (n \geq 39) ? 5 : (n \geq 7) ? 3 : (n - 1)/2;$$

b) With  $m$  as the larger of  $s+1$  and  $n$ , transform each element of the range (possibly more than once): iteratively for  $k = 0, \dots, m - 1$ , **calculate values**

$$r_1 = 1664525 \cdot T(\text{begin}[k] \text{ xor } \text{begin}[k+p] \text{ xor } \text{begin}[k-1])$$

$$r_2 = r_1 + \begin{cases} s & , k = 0 \\ k \bmod n + v[k-1] & , 0 < k \leq s \\ k \bmod n & , s < k \end{cases}$$

**and, in order, increment  $begin[k+p]$  by  $r_1$ , increment  $begin[k+q]$  by  $r_2$ , and set  $begin[k]$  to  $r_2$ .**

~~$$\text{begin}[k] \text{ xor } (1664525 * T(\text{begin}[k-1])) + v[k] + (k \bmod s).$$~~

c) Transform each element of the range ~~one last~~ **three more times**, beginning where the previous step ended: iteratively for

~~$k = m \bmod n, \dots, n - 1, 0, \dots, (m - 1) \bmod n$~~   $k = m, \dots, m + n - 1$ , **calculate values**

$$r_3 = 1566083941 \cdot T(\text{begin}[k] + \text{begin}[k+p] + \text{begin}[k-1])$$

$$r_4 = r_3 - (k \bmod n)$$

and, in order, update `begin[k + p]` by xoring it with  $r_4$ , update `begin[k + q]` by xoring it with  $r_3$ , and set `begin[k]` to  $r_4$ .

~~`(begin[k] xor (1566083941 * T(begin[k-1]))) -- k.`~~

## 10 Issue 712: `seed_seq::size` no longer useful

**Section:** 26.4.7.1 [rand.util.seedseq] **Submitter:** Marc Paterno **Date:** 2007-08-25

**Discussion:** One of the motivations for incorporating `seed_seq::size()` was to simplify the wording in other parts of [rand]. As a side effect of resolving related issues, all such references to `seed_seq::size()` will have been removed. More importantly, the present specification is contradictory, as “The number of 32-bit units the object can deliver” is not the same as “the result of `v.size()`.”

**Proposed resolution:** In 26.4.7.1, delete from the class synopsis:

~~`size_t size() const;`~~

Also delete the entirety of paragraph 10:

~~`size_t size() const;`  
10 *Returns:* The number of 32-bit units the object can deliver, as if by returning the result of `v.size()`.~~

**Recommendations:** We recommend that `seed_seq::size()` be removed, as above.

## 11 Issue 655: Signature of `generate_canonical` not useful

**Section:** 26.4.7.2 [rand.util.canonical] **Submitter:** Daniel Krügler **Date:** 2007-03-08

**Discussion:** In 26.4.2 [rand.synopsis] we have the declaration

```
template<class RealType, class UniformRandomNumberGenerator, size_t bits>
result_type generate_canonical(UniformRandomNumberGenerator& g);
```

Besides the “`result_type`” issue (already recognized by Bo Persson at Sun, 11 Feb 2007 05:26:47 GMT in this group) it’s clear, that the template parameter order is not reasonably chosen: Obviously one always needs to specify all three parameters, although usually only two are required, namely the result type `RealType` and the wanted bits, because `UniformRandomNumberGenerator` can usually be deduced.

**Proposed resolution:** In the header `<random>` synopsis 26.4.2 [rand.synopsis] as well as in the corresponding function description in 26.4.7.2 [rand.util.canonical] 26.4.7.2 between para 2 and 3 change the declaration

```
template<class RealType, class UniformRandomNumberGenerator,
        size_t bits, class UniformRandomNumberGenerator>
RealType generate_canonical(UniformRandomNumberGenerator& g);
```

**Recommendations:** We concur with the above analysis of this relatively minor issue, and agree that the utility of `generate_canonical` would benefit from the proposed resolution.

## 12 Summary and conclusion

This paper has recommended resolutions to all known issues 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.

## 13 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.