

# Random\_shuffle() in the Copenhagen STL

Simon Thamdrup Jensen  
Vesterbrogade 127A, st. th.  
1620 Copenhagen  
Denmark  
thamdrup@visto.com

Copenhagen STL Report 2001-2, December 2000.

**Abstract.** Two algorithms for shuffling the elements in a sequence is explored, explained and compared. The first one is a sequential algorithm, which is implemented in most if not all exiting implementations of the STL. The second one is a parallel algorithm which in turn is transformed to a cache optimized version designed to run a conventional machine with hierarchical memory. The benchmarking on the implementations of the algorithms executed on such machines, has surprisingly showed that the original sequential algorithm was in fact the fastest. So the conclusion was to keep that implementation for `random_shuffle` in the Copenhagen STL.

**Key words:** Copenhagen STL, `random_shuffle`

## 1 Preface

This report documents my activities for the component `random_shuffle`. What's not documented here, is the work we've had in the team with defining the project standards. For about four months now, we have had weekly project meetings to discuss things like: how to use the repository (a CVS-system), our coding style, our documentation style and structure, how to get some automatic documentation (we use a Doxygen-system), how should the L<sup>A</sup>T<sub>E</sub>X -template for our reports look like (we use the style-files provided for the Nordic Journal of Computing, even there is still some errors when using them). For the time being most of these software engineering problems are nearly solved, but cooperative work is time consuming, so everything isn't working without problems yet.

Anyway, I have been (and am) in a hurry because of some very sharp deadlines, so I had to do my work with `random_shuffle`, even though the preconditions was not at all optimal.

## 2 Introduction

Taken from the Copenhagen STL website ([www.cphstl.dk](http://www.cphstl.dk)), the projects main purposes are:

- to study and analyse existing specifications for and implementations of STL to determine the best approaches to optimization,
- to design alternative/enhanced versions of individual STL components using standard algorithmic and performance engineering techniques, and
- to implement and document the new versions in C++.

This report documents these activities for the STL template function `random_shuffle` in the Copenhagen STL Project.

## 3 Specification according to the standard

The template function `random_shuffle()` is included by including the standard header `<algorithm>`.

### 3.1 Prototypes

There are two overloaded versions of `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);
```

### 3.2 Description

The effect of applying `random_shuffle(first, last)` is to shuffle the elements in the range  $[first, last)$ ; that is, one of the possible  $n!$  permutations is randomly selected, where  $n = last - first$ .

The two versions of the function differs in that the first uses an internal default random number generator, and the second uses a generic random number generator which is being passed as a function object.

### 3.3 Requirements on types

- For both versions of the function, the template parameter, `RandomAccessIterator`, have to be a model of Random Access Iterator.
- For the second version, the template parameter, `RandomNumberGenerator`, have to be a model of Random Number Generator. An object, `rand` of that type takes an argument,  $n$ , of type `iterator_traits<RandomAccessIterator>::difference_type`, and returns a radomly chosen value of the type `iterator_traits<RandomAccessIterator>::difference_type` in the interval  $[0, n)$ .
- Because of the previous item, a type requirement is therefore that the distance type of `RandomAccessIterator` is convertible to the argument of `RandomNumberGenerator`.

### 3.4 Preconditions

- $[first, last)$  is a valid range
- The number of elements,  $last - first$ , is less than `RandomNumberGenerators` maximum value.

### 3.5 Complexity

The standard states that exactly  $(last - first) - 1$  swaps are performed; that is, the complexity of the function has to be linear in the number of elements. In big-O notation:  $O(n)$ , where  $n = last - first$

## 4 Two different solutions

There are different ways to shuffle a sequence of elements—here two different solutions will be presented: a sequential and a parallel algorithm.

### 4.1 A sequential algorithm

The traditional solution to the problem is the algorithm described in [2, p. 145]. This algorithm is the most widely used in existing STL implementations including SGI.

#### 4.1.1 Abstract description

Here is a formal abstract description of how the sequential algorithm shuffles the  $n$  elements  $(e_0, e_1, \dots, e_{n-1})$ :

```
for  $i := 0$  to  $n - 1$ 
    swap  $e_i$  with a random selected  $e_j$  such that  $0 \leq j \leq i$ 
```

### 4.1.2 Complexity

The complexity of this algorithm is exactly what the standard requires—it performs  $n - 1$  swap operations and runs in  $O(n)$  time.

### 4.1.3 Proof of the probability

A very important property for the algorithm is that each ordering of the elements are equally likely. Theoretically there are  $n!$  different permutations, and the probability for each of them should therefore be  $\frac{1}{n!}$ .

PROOF. For each of the  $n - 1$  swap operations, the algorithm generates a random number,  $j$ , such that  $0 \leq j \leq i$ , where  $i$  is the number of operation. A permutation,  $\pi$ , generated by the algorithm can then be expressed as a sequence of random numbers:  $S_\pi = (j_1, j_2, \dots, j_{n-1})$  where  $0 \leq j_i \leq i$ .

The probability that the algorithm generates a specific permutation,  $\pi'$ , is then the probability that a specific sequence of random numbers,  $S_{\pi'}$ , is generated:

$$Pr[\pi = \pi'] = Pr[S_\pi = S_{\pi'}]$$

If the two sequences are equal, then each of their elements must also be:

$$= Pr \left[ \bigwedge_{1 \leq i \leq n-1} S_{\pi(i)} = S_{\pi'(i)} \right]$$

These probabilities can be multiplied as well:

$$= \prod_{1 \leq i \leq n-1} Pr [S_{\pi(i)} = S_{\pi'(i)}]$$

A random number,  $j_i$  at position  $i$  in the sequence must be in the range  $0 \leq j_i \leq i$ . There is therefore  $i + 1$  different possible numbers in each case, and the probability that one specific comes out is then  $\frac{1}{i+1}$ :

$$\begin{aligned} &= \prod_{1 \leq i \leq n-1} \frac{1}{i+1} \\ &= \prod_{2 \leq i \leq n} \frac{1}{i} \\ &= \frac{1}{n!} \square \end{aligned}$$

□

#### 4.1.4 Implementation

The implementation of this algorithm is straightforward, and is the version already included in existing implementations of the STL—also SGI’s. The iterator category have to be random access iterator because the algorithm must access the elements in random order. Well strictly speaking: it is possible to implement a version taking bidirectional iterators, and then using them to traverse back and forth in the sequence for every “random access”, but that would be  $O(n^2)$  time complexity, which is far to much compared to the requirements of the C++ standard.

The solution is then just to rely on the SGI implementation (see the helper function `knuth_perm` in A).

## 4.2 A parallel algorithm

Another algorithm for shuffling a sequence of elements is proposed by Peter Sanders in [3]. Sanders addresses the problem of random memory accesses and tries to minimize the number of page and cache faults.

In Knuth’s algorithm (see above) the problem is, that it is inherently sequential, and reads/writes each element twice; that is, it makes  $2n$  memory accesses, where  $n$  is the number of elements to shuffle. For large  $n$  this will lead to many page faults on external memory and to cache misses on hierarchical memory.

Sanders approach is to share the shuffling between a number of parallel processes with local memory—such a thing is called a *processing element* (PE) in the literature on parallel algorithms. The number of processing elements should be big enough so that they can each shuffle their part without page or cache faults.

### 4.2.1 Abstract description

Here is an abstract description (almost as Peter Sanders puts it) of how the parallel algorithm shuffles the  $n$  elements  $(e_0, e_1, \dots, e_{n-1})$ :

Number each of the  $P$  processing elements with  $0 \leq i_{PE} < P$ . Initially each PE contains  $\frac{n}{P}$  elements such that PE  $i$  contains elements  $e_j$  for  $in/P \leq j < (i + 1)n/P$ .

```
for each PE dopar
  send each element to a random PE
  store incoming elements in  $[t_0, \dots, t_{k-1}]$ 
  randomly permute  $[t_0, \dots, t_{k-1}]$  locally
   $\Delta := \sum_{i < i_{PE}} k_i$ , where  $k_i$  is  $k$  at PE  $i$ 
  put  $t_j$  into PE  $\lfloor P(\Delta + j)/n \rfloor$  at position  $\Delta - (n/P)\lfloor P(\Delta + j)/n \rfloor + j$ 
```

This idea is then transformed into an algorithm that works on a conventional machine: The permutation is here performed in *buckets* of size less than the size of the memory (which can be some cache level or external memory). A loosely (but easy to understand) description of this transformed algorithm is:

- Randomly distribute the elements to a number of buckets—manageable in fast memory.
- Permute the elements in each bucket (with the sequential algorithm)
- Send the elements back to output in the order they occur in each bucket.

#### 4.2.2 Complexity

Peter Sanders shows that the time needed for each of the processing elements is:

$$T_{par} \in O(n/P + \log P)$$

provided that the network communication is optimal. In this project we are more interested in the complexity for the algorithm when it is designed to run on a conventional machine.

Let us assume that we have enough fast memory to store  $M$  elements. Because we don't want page faults or cache misses to happen, then we need at least  $P = n/M$  buckets to shuffle the elements.

So first the elements have to be distributed to a random bucket. This takes time:

$$T_{dist} \in O(n)$$

provided that generating the random number takes constant time. Then we permute each of the  $P$  buckets with the sequential algorithm. This takes time:

$$T_{perm} \in P O(n/P) = O(n)$$

Then there is the last step of sending the elements back to output in the order they occur in each bucket. This is also a distribution problem which takes time  $T_{dist}$ . Because the algorithm's three parts have a time complexity of  $O(n)$  each, then the time complexity of the whole algorithm is also  $O(n)$ .

#### 4.2.3 Proof for probability

The important question now is: is every possible permutation generated with probability  $1/n!$ ? In fact it is, and the proof goes as follows:

PROOF. If  $\pi'$  is a specific permutation and  $\pi$  is the permutation generated, then the job is to show that the probability of these two permutations being equal is  $1/n!$ .

$$\begin{aligned} Pr[\pi = \pi'] &= Pr \left[ \bigwedge_{i < n} \pi(i) = \pi'(i) \right] \\ &= \prod_{i < n} Pr[\pi(i) = \pi'(i)] \end{aligned}$$

$1/n!$  can be written as  $\prod_{i < n} 1/(n - i)$ , and therefore we only have to prove that

$$Pr [\pi(i) = \pi'(i)] = \frac{1}{n-i}$$

If we take a closer look at the algorithm, we see that  $\pi(i) = \pi'(i)$  if and only if  $e_i$  is shuffled to position  $i$ . This happens only if these two conditions both holds:

1.  $e_i$  is sent to bucket  $l$ , where  $l$  is chosen such that

$$\sum_{j < l} k_j < i \leq \sum_{j \leq l} k_j$$

(the sum  $\sum_{j < l} k_j$  is also denoted  $\Delta_l$ ; the number of elements in buckets  $j$  for  $j < l$ )

2. After the distribution to bucket  $l$ , the element  $e_i$  is then sent to position  $i - \Delta_l$  in this bucket.

At the time (1) where  $e_i$  is to be sent to bucket  $l$ , there are still  $n-i$  elements to be sent, and  $k'$  of them are eventually placed in bucket  $l$ . The probability that  $e_i$  is one of them is  $k'/(n-i)$ .

When the distribution is over, we advance to the local permutation (2), where  $e_i$  is shuffled to the local position  $i - \Delta_l$ . The probability for this to happen is  $1/k'$ .

The probability that  $\pi(i) = \pi'(i)$  is then the product of the probabilities for (1) and for (2),

$$\frac{k'}{n-i} \frac{1}{k'} = \frac{1}{n-i} \quad \square$$

#### 4.2.4 Implementation

Peter Sanders has made an implementation of the algorithm available at his web-site (see [4]). His implementation is a C-style program, which I in turn have rewritten to fit the C++ standards requirements for `random_shuffle` (see A).

One of the main factors for this algorithm to perform well, is the way the random numbers are generated. If the user supplies a random function, then it's not my problem, but when she don't—then I'll have to choose one, that does a good job: That is, it generates random numbers with a statistically equal distribution, and it does it fast.

It is generally known, that the standard c function `rand()` doesn't do a good job in the above sense. In the Copenhagen STL team we decided to find another better random function, and by the time of writing, the chosen one is ISAAC's random number generator (see [1]).

Concerning exception safety, the standard requires the *basic guarantee* for all parts of the standard library. The basic guarantee states, that there will not be leaks of resources or violations of container invariants in the face of exceptions.

This guarantee is based on the requirement that destructors never throws. In my case that's especially important for the generic random generator and for the iterators. There are also some other general rules for using the STL including, that iterators and their ranges are valid.

To make sure that the implementation meets the basic guarantee it only uses local automatically destructed objects (an exception safe way to avoid resource leaks). But if the algorithm is used in threads, the iterators might be violated, because the implementation isn't made thread safe.

If this implementation should be made thread safe, then the hole input sequence would have to be copied before shuffling it. That would be memory and especially time consuming ( $O(n)$  extra space and time). So if thread safety if a must, the user will have to take care her self, i.e. make a wrapper class with a local copy of the sequence.

In the Copenhagen STL team we had some discussions about to what degree we should use the loop unrolling technique for best pipeline performance, and we decided to have the compiler do the work. The philosophy is then, than if speed is required, one should first acquire a good compiler. The reason for this is, that we then can present more readable source code.

Because the success of this algorithm highly depends on the actual sizes of the memory on which it is run, an implementation also must consider how to extract those properties automatically. But this problem is at the moment not solved yet, so the values are just hard coded in the source code

## 5 Benchmarks

To compare the performance of the two implementations, I carried some time benchmarks out. Both of the implementations were given vectors of increasing sizes from  $2^{16}$  to  $2^{27}$  elements, and the result can be seen in Fig. 1. The upper most curve is benchmarks from the parallel implementation (on the graph called `cphstl::random_shuffle`), and the other is from the sequential algorithm which is used in the SGI STL (on the graph called `std::random_shuffle`).

## 6 The solution chosen

The benchmarks shows that the sequential algorithm is the fastest—at least on that machine (a PIII Coppermine) at that moment. I have also tried to run my benchmarking on other machines, but with similar results. So the solution for Copenhagen STL must for now be to include the implementation of the sequential algorithm which is already in SGI's STL.

# Bechmarks random\_shuffle

- comparing the SGI and CPHSTL versions

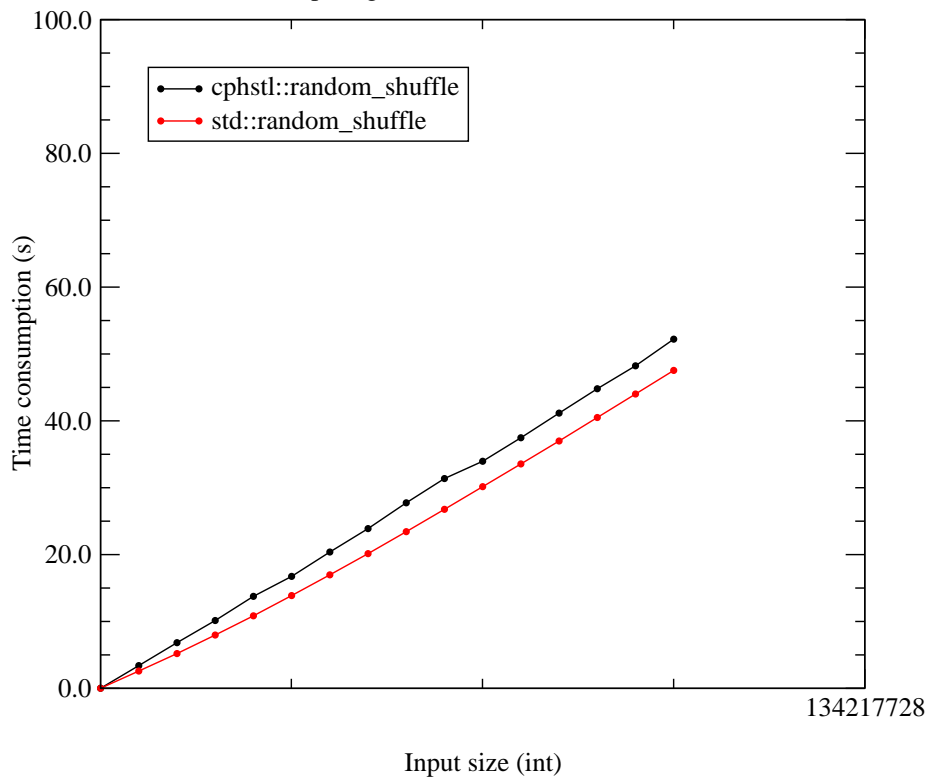


Figure 1: Benchmarks generated with Grace

## 7 Conclusion

The presented algorithms are interesting from a theoretical and a pedagogical point of view because they show an example on how a problem can be solved with both a sequential and a parallel solution. Moreover it's interesting to see how the parallel algorithm can be transformed into a solution designed for a conventional machine with hierarchical memory.

The parallel algorithm was implemented in hope for better performance compared to existing STL implementations, but I was not able to show that it in fact was! So the conclusion is to keep the existing implementation, but also to continue the experimentations with the theoretically faster algorithm—in hope to be able to present a solution in the future, which is also faster in practice.

## A Source code for the parallel algorithm

```
/*! \file
 * \brief This file defines the function
 * <code>random_shuffle</code>.
 *
 * <h3>
 * Original author
 * </h3>
 *
 * Simon Thamdrup Jensen <thamdrup@visto.com>,
 * October - December 2000
 *
 * <h3>
 * Later modifications
 * </h3>
 *
 *
 *
 */

#include <algorithm> /* iter_swap; max; */
#include <iterator> /* iterator traits and iterator tags */
#include <vector>
#include <cmath> /* log */
#include <cstddef> /* ptrdiff_t */
//#include "functional.h"
#include <functional> /* random_number */

namespace cphstl {
```

```

namespace helper {

class random_generator {
    cphstl::random_number<> randnum;
    cphstl::ISAAC_INT max_value;
public:
    random_generator() {
        /* Set max_value to the biggest possible number;
           that is, 0xFF in all of the
           sizeof(ISAAC_INT) bytes
        */
        unsigned int size = sizeof(cphstl::ISAAC_INT);
        max_value = 0xFF;
        for (size_t i = size; i>1; --i) {
            max_value = (max_value<< 8) | 0xFF;
        }
    }
    ptrdiff_t operator() (ptrdiff_t max) {
        /* return a random number, r, in the
           range 0 <= r < max
        */
        return (ptrdiff_t)
            ( max *
              ( static_cast<double>( randnum.rand() ) /
                static_cast<double>(max_value) ) );
    }
};

/* D. Knuths algorithm is used as part of the
   CPHSTL algorithm, ie. it's a helper fuction.
*/
template<typename iterator,
         typename random_number_generator>
void knuth_perm(iterator first, iterator last,
               random_number_generator& random_function) {
    if (first == last) {
        return;
    }
    for (iterator i = first + 1; i != last ; ++i) {
        std::iter_swap(i,
                      first + random_function(i-first+1));
    }
}
} // namespace helper

```

```

/***** And here goes the cphstl-version *****/

/* These memory constants should be defined in some
   header file. In this file indeed we should have
   constants for different memory hierachies

   #include <memory_constants> or something like that
*/
/*
   Fast memory available for permuting
   (should not be too tight)
   1st level cache size for small inputs
   2nd level cache size for medium inputs
   RAM size for large inputs
*/
static const int M = 256 * 1024; //i.e. a PIII (Coppermine)

/*
   Blocksize of slow memory
   cache line size for small inputs
   page size for large inputs
*/
static const int B = 256/8; //i.e. a PIII (Cobbermine)

/*! \ingroup algorithm
   * \brief Function <code>random_function</code>
   *
   * <h3>
   * Input
   * </h3>
   *
   * A sequence given by its <code>first</code> and
   * <code>last</code> positions, and an optional
   * random number generator <code>random_function</code>.
   *
   * <h3>
   * Effect
   * </h3>
   *
   * Rearranges the elements in the range
   * <code>[first, last)</code>; that is, it randomly
   * picks one of the N! possible permutations,
   * where N is <code>last - first</code>.
   * If a random number generator is passed to the

```

```

* function via the argument random_function,
* then that function will be used in the generation of
* the random permutations. Otherwise a default
* random function is used.
*
* 

### * Requirements *


*
* 


* - The type random_access_iterator must
* have the capabilities of an random access iterator.
*

* - The type random_number_generator must
* be a model of random number generator.
*

* - The operator() of
* unary_functor must take an integral
* value, n, of type difference_type of
* the iterator. And it must return a random number
* greater than or equal to zero and less than n.
*

* 

*
* 

### * Preconditions *


*
* 


* - [first, last) must be
* a valid range.
*

* - The number of elements, last-first,
* must be less than the random number generators
* maximum value.
*

* 

*

```

```

* <h3>
* Safety
* </h3>
*
* This function is exception safe to the degree of
* the so called basic guarantee; that is,
* it doesn't leak resources and it doesn't invalidate
* the iterators. The guarantee is given under
* the preconditions described above.
*
* The function is not thread safe, and threads
* should therefore keep their own copies of the
* data in the sequence.
*
* <h3>
* Implementation
* </h3>
*
* The function appears in two versions; one that takes
* a random number generator, and one that doesn't.
* The last one uses a default internal
* random number generator based on a fast
* random function from ISAAC.
*
* Compared to the SGI STL implementation the
* implementation here uses a cache optimized algorithm,
* which unformal can be described as follows:
* It puts the elements into random buckets
* (of sequential memory words) of sizes manegable
* in cache, and does the permutations local in each
* bucket. Then it concatenates the buckets back to
* the sequence. (The algorithm is proposed in
* Peter Sanders. 1998. Random Permutations on
* distributed, external and hierarchical memory.
* Information Processing Letters 67, 305-309.)
*
* This implementation does not do loop unrolling
* explicitly, but it is left for the
* compiler. The philosophy is then, than if speed is
* required, one should first acquire a good compiler.
* The reson for this is, that we then can present a
* more readable source code.
*
* <h3>
* Performance
* </h3>

```

```

*
* <ul>
* <li>
* At most each of the N elements are copied two times
* and swaped once; that is, it runs in  $O(N)$ , where N
* is last - first.
* </li>
*
* <li>
* In this implementation, the performance is optimized
* for a cahce of 256 Kb and line size 257 bytes (32 Kb),
* but it is the intention that these memory
* properties should be automatically retrieved for the
* actual system, the CPHSTL components are executed on.
* </li>
*
* <li>
* With full optimization (-O6 g++ dvalin) the function
* was almost as fast as the version available in the
* SGI STL, but unfortunatly not faster :-(.
* So I did not succeed to prove that this theoretically
* better solution for big N's in fact is faster.
* More experimentation has to be done in the future.
* </li>
* </ul>
*
* <h3>
* Known Bugs
* </h3>
*
* This version still relies on std::..
*
* <h3>
* Test Program
* </h3>
*
*/

// \include random_shuffle_test.cpp

template<typename random_access_iterator,
         typename random_number_generator>
void random_shuffle(random_access_iterator first,
                   random_access_iterator last,
                   random_number_generator& random_function) {

```

```

//Iterator traits for the passed iterator:
typedef typename
    std::iterator_traits<random_access_iterator>
        ::value_type value_type;
typedef typename
    std::iterator_traits<random_access_iterator>
        ::difference_type difference_type;

//TODO:
/* typecheck that the iterator is a model of
    RandomAccessIterator; maby with the SGI-macro
*/

/* Number of elements.
    (I asume that difference_type is always integral).
*/
difference_type n = last-first+1;

/* Make buckets as large as possible as long as one
    bucket fits into fast memory.
    p, a power of 2, is the number of buckets
*/
const size_t logp =
    std::max(0, (int)(log(n *
        sizeof(value_type) /
        (double)M
        ) / log(2.0) + 1e-5));
const size_t p = 1 << logp;

if (p == 1) {
    helper::knuth_perm(first, last, random_function);
    return;
}

/* Make a buffer to store the values of the buckets in.
    It is crucial that the vector used here stores
    it's elements sequential, but the C++ standard
    requires that it does.
*/
size_t buffer_size =
    (size_t)(2.1 * n + sqrt(20 *
        n * (20 + log((double) n))));
std::vector<value_type> buffer;
buffer.reserve(buffer_size);

/* Make a vector to keep track of where each

```

```

    bucket is in the buffer.
    */
typedef typename
    vector<value_type>::iterator buffer_iterator;
std::vector<buffer_iterator> bucket;
bucket.reserve(p);

/* Place the buckets inside the buffer, with as much
   spacing as possible, but leave
   n Elements free at the end so that even in the worst
   case there is no segmentation violation
*/
const size_t bucket_size = (buffer_size - n) / p;
register size_t i;
for (i = 0; i < p; ++i) {
    bucket[i] = buffer.begin() + i * bucket_size;
}

/* First critical loop:
   put elements into random buckets
*/
buffer_iterator put_pos;
random_access_iterator pos;
for (pos = first; pos <= last; ++pos) {
    i = random_function(p);
    put_pos = bucket[i];
    *put_pos = *pos;
    put_pos = put_pos + 1;
    bucket[i] = put_pos;
}

/* Check for bucket overflow.
   That is, if a bucket iterator points outside
   it's own area in the buffer, then we fall back
   on the sequential algorithm.
*/
for (i = 0; i < p; ++i) {
    if (bucket[i] >
        buffer.begin() + (i + 1) * bucket_size) {
        helper::knuth_perm(first, last, random_function);
        return;
    }
}

/* Second critical loop:
   Copy the buckets back to input and permute them there.

```

```

        The copying is done, so that the elements
        line up without any spacing between them.
    */
    pos = first;
    buffer_iterator this_bucket;
    typename std::iterator_traits<buffer_iterator>
        ::difference_type elements_in_bucket;
    for (i = 0; i < p; ++i) {
        this_bucket = buffer.begin() + i * bucket_size;
        // bucket[i] points to the first element in bucket i+1
        elements_in_bucket = bucket[i] - this_bucket;

        // We first copy to input and then we permute
        std::copy(this_bucket,
                 this_bucket + elements_in_bucket - 1, pos);
        helper::knuth_perm(pos,
                          pos + elements_in_bucket - 1,
                          random_function);
        pos += elements_in_bucket;
    }
} // random_shuffle()

template<typename random_access_iterator>
void random_shuffle(random_access_iterator first,
                  random_access_iterator last) {
    cphstl::helper::random_generator random_function_object;
    random_shuffle(first, last, random_function_object);
} // random_shuffle()

} // namespace cphstl

```

## References

- [1] ISAAC, Isaac: a fast cryptographic random number generator., <http://www.burtleburtle.net/bob/rand/isaacafa.html> (2000).
- [2] D.E. Knuth, *The Art of Computer Programming—Seminumerical algorithms, Vol. 2*, 2nd Edition, Addison-Wesley (1981).
- [3] P. Sanders, Random permutations on distributed, external and hierarchical memory, *Information Processing Letters* **67** (1998), 305–309.

- [4] P. Sanders, Some of peter sanders's programs. random permutations.,  
*<http://www.mpi-sb.mpg.de/~sanders/programs/randperm>* (2000).