

Text-line Random Shuffling Program

David R. Musser

October 12, 2001

Contents

1 Introduction	1
2 The Program	2
2.1 Overview	2
2.2 Randomization	2
2.3 Inputting the lines	4
2.4 Outputting the lines	6
3 Makefile and Sample Input	6

1 Introduction

This little program reads a sequence of text lines and writes them in a scrambled (randomized) order. Given any sequence of items formatted one item per line, this program can be used to produce the sequence permuted in a random order. The program takes the text-line input from the standard input stream and produces the output lines on the standard output stream. It also takes one input from the command line and uses it as an integer seed for a random-number generator, so that by running the program with a different values of this command line parameter, one can obtain different randomized orders of the given line sequence. A “line” is considered to be any sequence of characters other than end-of-line, followed by (and including) an end-of-line character.

The program is written in the C++ language and makes use of several C++ standard library components from the `iostream`, `string`, `vector`, and `algorithm` sections of the standard. It may be useful to read the code as a short, simple illustration of some of the advantages of using such standard library components. The presentation of the code is also a simple illustration of “literate programming” as defined by D. E. Knuth [Knuth84]. The main idea behind literate programming is that many programs are more often read by humans than executed, so it’s vitally important to make them readable and to ensure that documentation remains consistent with the code. Another key idea is that presenting programs in segments arranged in a logical order (rather than the

order required by the compiler) actually helps the programmer to design good programs in the first place. So one should *not* first write the code and then start reorganizing it for purposes of documentation; instead it's much better to code and document at the same time.

Code is presented in “parts” numbered according to the page number on which they appear, with parts on the same page distinguished by appending a letter to the number. The order in which the parts appear in the documentation does not have to be the same order in which they must appear in the code source file(s) according to the programming language rules. This form of presentation and code construction is supported by Briggs' Nuweb tool [Briggs], which generates both the code file and the documentation file directly from a single Nuweb source file. Any number of auxiliary files, such as makefiles and test data files, can also be packaged in the Nuweb source file, as is illustrated in the last section of this document.

The presentation in this document assumes familiarity with major C++ language features and standard library components.

2 The Program

2.1 Overview

A top-down view of the program structure is as follows:

```
"rand.cpp" 2 ≡
  <Include standard header files 4a>
  <Make names of standard components directly available 4b>
  <Define a type, line, for processing text-lines 5a>
  <Define a random number generator in form required by random shuffle 3b>

int main(int argc, char* argv[])
{
  <Declare a vector, lines, into which to read the input sequence 5d>
  <Seed the random number generator using command line argument 1 3c>
  <Copy lines from standard input stream, until end-of-file, to lines vector 5e>
  <Randomize order of lines vector elements 3a>
  <Copy lines from lines vector to standard output stream 6a>
  return 0;
}
```

2.2 Randomization

The heart of the program is the randomizing step:

⟨Randomize order of lines vector elements 3a⟩ ≡

```
random_shuffle(lines.begin(), lines.end(), randgen1);
```

Used in part 2.

This uses a standard library component, `random_shuffle`, a generic function (from the `<algorithms>` header), whose first two arguments are random access iterators delimiting a sequence of values, and whose third parameter is a random number generator function object. For the third parameter `random_shuffle` needs a function, encapsulated as a function object, that takes an integer argument n and returns a value in the range $[0, n)$. We program this function object, `randgen1`, in terms of `lrand48`, a function available in the library of some C/C++ systems. The argument and return types of the encapsulated function need to be determined based on the type of container `random_shuffle` is applied to, so we make the container type a template parameter.

⟨Define a random number generator in form required by random shuffle 3b⟩ ≡

```
template <class Container>
struct randgen {
    typedef typename Container::difference_type argument_type;
    typedef typename Container::difference_type result_type;
    result_type operator()(argument_type n)
    {
        return lrand48() % n;
    }
};

randgen<vector<line> > randgen1;
```

Used in part 2.

Whenever we use a random number generating function, we have to provide for “seeding” it, so that by using different seeds different runs of the program will produce different (apparently random) results. The companion seeding function to `lrand48` is `srand48`.

⟨Seed the random number generator using command line argument 1 3c⟩ ≡

```
if (argc == 2)
    srand48(atoi(argv[1]));
else {
    srand48(7);
    cerr << "*****Rerun with one command line argument, an integer, "
         << "to get a different scrambling of the output." << endl;
}
```

Used in part 2.

Instead of `lrand48` and `srand48` it would seem simpler and more portable to use `rand` and `srand`, the standard C library random number function and seeding function. But `rand` is a poor generator and should only be used for test pur-

poses, if at all. Unfortunately, no new random number generator requirement was added to the C++ standard. When compiling programs such as this one with a C/C++ library that doesn't have `lrand48`, check for other generators that might be available before resorting to `rand`. If `rand` and `srand` must be used, they can be conveniently substituted into the program with preprocessor commands since each of these functions has the same interface as its counterpart: either add the lines

```
#define lrand48 rand
#define srand48 srand
```

at the beginning of the program or achieve the same effect by inserting the definitions on the compilation command line; e.g.,

```
bcc32 -Dlrand48=rand -Dsrand48=srand rand.cpp
```

2.3 Inputting the lines

The rest of the program code is concerned mainly with input of text lines into the vector and output from it. We first include the `iostream`, `string`, `vector`, and `algorithm` library headers.

```
(Include standard header files 4a) ≡
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
```

Used in part 2.

Identifiers used in standard headers are introduced within a *namespace* called `std`. To make them more conveniently available, the `using namespace` construct is useful.

```
(Make names of standard components directly available 4b) ≡
using namespace std;
```

Used in part 2.

For its operations on text lines, the standard string class provides most of what the program needs. However, the input facility of the string class provided by its overloading of `operator>>` is not exactly what is needed, since it stops absorbing characters into the string it is reading from an input stream when it reads any whitespace character or certain special characters. What we need here is to include all characters from the present input stream position up to and including the next end-of-line character (denoted by `\n` in C++ source code). We can get the combination we need—input of whole text lines with `operator>>` and standard string behavior for all other operations—by first deriving a new class, `line`, from `string`, and then changing the meaning of `operator>>`:

⟨Define a type, `line`, for processing text-lines 5a) ≡

```
class line : public string { };
```

⟨Define input operator to gather all characters through the next end-of-line 5b) ≡

Used in part 2.

We overload `operator>>` on `line` type objects to have the behavior we need for this program:

⟨Define input operator to gather all characters through the next end-of-line 5b) ≡

```
istream& operator>>(istream& i, line& s) {  
    s.clear();  
    ⟨Get characters from i and push them on the back of s 5c) ≡  
    return i;  
}
```

Used in part 5a.

The standard library function `getline` could be used here, but it requires an array of some predetermined size for storing the characters. We can avoid having any arbitrary limit on the length of lines by programming the scanning and storing of the characters directly using `get` and `push_back`.

⟨Get characters from i and push them on the back of s 5c) ≡

```
while (true) {  
    char c;  
    i.get(c);  
    if (i.eof())  
        break;  
    s.push_back(c);  
    if (c == '\n')  
        break;  
}
```

Used in part 5b.

Having defined type `line`, we can now prepare a vector to hold items of the type so that they can be shuffled.

⟨Declare a vector, `lines`, into which to read the input sequence 5d) ≡

```
vector<line> lines;
```

Used in part 2.

We can easily read the lines into the `lines` vector using the generic copy algorithm in combination with `istream` iterators.

⟨Copy lines from standard input stream, until end-of-file, to lines vector 5e) ≡

```
typedef istream_iterator<line> line_input;  
copy(line_input(cin), line_input(), back_inserter(lines));
```

Used in part 2.

It is here that `operator>>` is actually used, since `istream_iterator<line>` uses that operator to supply `line` values to the `copy` algorithm. This call of `copy` places the result into `lines` with the aid of `back_inserter`, an iterator adaptor that converts assignment operations into uses of the `push_back` operation of its container argument.

With the lines now in the `lines` vector, we can shuffle them with the generic `random_shuffle` algorithm, as already shown.

2.4 Outputting the lines

Finally, to write the lines to the output stream, we can again use the generic `copy` algorithm, this time in combination with an ostream iterator.

```
(Copy lines from lines vector to standard output stream 6a) ≡
    typedef ostream_iterator<line> line_output;
    copy(lines.begin(), lines.end(), line_output(cout, ""));
```

Used in part 2.

Symmetrically to the input case, `operator<<` is used here: the `copy` algorithm and `ostream_iterator<line>` working together use that operator to place `line` values on the output stream, `cout`. The `ostream_iterator<line>` constructor takes two arguments, an ostream to write the output to (`cout` in this case), and a string to use to separate consecutive values (the empty string `""` in this case).

3 Makefile and Sample Input

The following Unix makefile provides for compiling the program, executing it with sample data, and producing its formatted documentation file in either Postscript or PDF.

```
"Makefile" 6b ≡
    rand-w: rand.w
            pdfnuweb rand

    rand-comp: rand-w
              bcc32 -Dlrnd48=rand -Dsrand48=srand rand.cpp

    rand-run: rand-comp
              ./rand <sample.txt >scrambled1.txt 37
              ./rand <sample.txt >scrambled2.txt 9

    rand-pdf: rand-w
              pdflatex rand
              pdfnuweb rand
              pdflatex rand

    rand-ps: rand.w
```

```
nuweb rand
latex rand
nuweb rand
latex rand
dvips -o rand.ps rand
```

"sample.txt" 7 ≡

```
A.J. Perlis
Maurice V. Wilkes
Richard Hamming
Marvin Minsky
J.H. Wilkinson
John McCarthy
E.W. Dijkstra
Charles W. Bachman
Donald E. Knuth
Allen Newell
Herbert A. Simon
Michael O. Rabin
Dana S. Scott
John Backus
Robert W. Floyd
Kenneth E. Iverson
C. Antony R. Hoare
Edgar F. Codd
Stephen A. Cook
Ken Thompson
Dennis M. Ritchie
Niklaus Wirth
Richard M. Karp
John Hopcroft
Robert Tarjan
John Cocke
Ivan Sutherland
William (Velvel) Kahan
Fernando J. Corbato'
Robin Milner
Butler W. Lampson
Juris Hartmanis
Richard E. Stearns
Edward Feigenbaum
Raj Reddy
Manuel Blum
Amir Pnueli
Douglas Engelbart
James Gray
Frederick P. Brooks, Jr.
```

References

- [Briggs] P. Briggs, *Nuweb, a simple literate programming tool*, Version 0.87, 1989. [1](#)
- [Knuth84] D.E. Knuth, Literate programming. *Computer Journal* 27 (1984), 97–111. [1](#)