Text-line Random Shuffling Program

David R. Musser

April 25, 2005

Abstract

This short document presents a program that takes as input any sequence of items formatted one item per line and outputs the sequence permuted in a random order. The program is written in the C++ language and makes use of several C++ standard library components; it is a succinct example of the advantages of using such components rather than relying solely on facilities of the programming language itself. It is furthermore a simple illustration of the literate programming style of simultaneous development of code and its documentation.

Contents

1 Introduction **1**

2 The Program **3**
2.1 Overview 3
2.2 Randomization 3
2.3 Inputting the lines 5
2.4 Outputting the lines 7

3 Makefile and Sample Input **7**

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, iterator, 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 [2]. 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 the programming language rules. This form of presentation and code construction is supported by the the literate programming tool Nuweb, which generates both the code file and the documentation file directly from a single Nuweb source file.[1] 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.

[1] Nuweb was originally developed by P. Briggs [1] and has been extended and modified by others, including J. Ramsdell, M. Mengel, and D. Musser. Pdfnuweb is an extension of Nuweb, developed by R. Loos, that adds hyperlinking to the code part definitions and uses in the documents it produces, in the form recognized by the hyperref package for pdflatex. The Nuweb version and Pdfnuweb versions used in producing this document are available online [3].

2 The Program

2.1 Overview

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

"shuffle-lines.cpp" 3a ≡

```
⟨Include standard header files 5a⟩
⟨Make names of standard components directly available 5b⟩
⟨Define a type, line, for processing text-lines 5c⟩
⟨Define a random number generator in form required by random shuffle 4a⟩

int main(int argc, char* argv[])
{
  ⟨Declare a vector, lines, into which to read the input sequence 6c⟩
  ⟨Seed the random number generator using command line argument 1 4b⟩
  ⟨Copy lines from standard input stream, until end-of-file, to lines vector 6d⟩
  ⟨Randomize order of lines vector elements 3b⟩
  ⟨Copy lines from lines vector to standard output stream 7a⟩
  return 0;
}
```

2.2 Randomization

The heart of the program is the randomizing step:

⟨Randomize order of lines vector elements 3b⟩ ≡

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

Used in part 3a.

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 4a⟩ ≡

```
template <typename 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 3a.

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 4b⟩ ≡

```
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 3a.

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 purposes, 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` must be used, one can simply replace the identifier `lrand48` by `rand` and `srand48` by `srand` since each of these functions has the same interface as its counterpart.

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 needed library headers.

⟨Include standard header files 5a⟩ ≡

```
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include <algorithm>
```

Used in part 3a.

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 5b⟩ ≡

```
using namespace std;
```

Used in part 3a.

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 5c⟩ ≡

```
class line : public string { };
⟨Define input operator to gather all characters through the next end-of-line 6a⟩
```

Used in part 3a.

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 6a⟩ ≡

```
istream& operator>>(istream& i, line& s) {
  s.clear();
  ⟨Get characters from i and push them on the back of s 6b⟩
  return i;
}
```

Used in part 5c.

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 6b⟩ ≡

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

Used in part 6a.

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 6c⟩ ≡

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

Used in part 3a.

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 6d⟩ ≡

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

Used in part 3a.

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 7a⟩ ≡

```
typedef ostream_iterator<line> line_output;
copy(lines.begin(), lines.end(), line_output(cout, ""));
```

Used in part 3a.

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 makefile provides for compiling the program, executing it with sample data, and producing its formatted documentation file in PDF. The PDF produced with the `screen-doc` target will implement cross references as hyperlinks, including not only those that LaTeX generates (such as between citations and the references cited) but also those that Nuweb generates between code part definitions and their uses in other parts. These hyperlinks are highlighted on the screen in various colors, some of which do not show up well if the document is sent to a black-and-white printer, hence the makefile also provides a `print-doc` target that produces PDF without hyperlinks.

`"Makefile"` 7b ≡

```
SOURCE = shuffle-lines
CC = g++3
```

```
compile: ${SOURCE}.w
        nuweb ${SOURCE}
        ${CC} -O ${SOURCE}.cpp -o ${SOURCE}

test: compile
        ./${SOURCE} <sample-input.txt >scrambled1.txt 37
        ./${SOURCE} <sample-input.txt >scrambled2.txt 9

screen-doc: ${SOURCE}.w
        pdfnuweb ${SOURCE}
        pdflatex ${SOURCE}
        bibtex ${SOURCE}
        pdfnuweb ${SOURCE}
        pdflatex ${SOURCE}

print-doc: ${SOURCE}.w
        nuweb ${SOURCE}
        latex ${SOURCE}
        bibtex ${SOURCE}
        nuweb ${SOURCE}
        latex ${SOURCE}
        dvipdfm ${SOURCE}

# Check the following carefully to see that the only files deleted are
# those generated by Nuweb and other tools
clean:
        rm *.o *.exe core *.tex *.pdf *.dvi *.aux \
           *.log *.bbl *.blg *.brf *.cpp *.out *.toc \
           *.txt *.bib
```

The following sample text file (a list of Turing Award winners) is used in the tests of the program. With this input, the output is a pseudo-random ordering of the names in the file.

"sample-input.txt" 8 ≡

```
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

[1] Preston Briggs. *Nuweb, a simple literate programming tool.* `http://www.literateprogramming.com/fdownload.html`.

[2] D. E. Knuth. Literate programming. *The Computer Journal*, 27:97–111, 1984.

[3] David R. Musser. *Literate generic programming.* `http://www.cs.rpi.edu/~musser/gp/literate.html`.