by Jon Bentley

with Special Guest Oyster Don Knuth

programming pearls

LITERATE PROGRAMMING

When was the last time you spent a pleasant evening in a comfortable chair, reading a good program? I don't mean the slick subroutine you wrote last summer, nor even the big system you have to modify next week. I'm talking about cuddling up with a classic, and starting to read on page one. Sure, you may spend more time studying this elegant routine or worrying about that questionable decision, and everybody skims over a few parts they find boring. But let's get back to the question: when was the last time you read an excellent program?

Until recently, my answer to that question was, "Never." I'm ashamed of that. I wouldn't have much respect for an aeronautical engineer who had never admired a superb airplane, nor for a structural engineer who had never studied a beautiful bridge. Yet I, like most programmers, was in roughly that position with respect to programs. That's tragic, because good writing requires good reading—you can't write a novel if you've never read one. But the fault doesn't rest entirely with us programmers: most programs are written to be executed, a few are written to be maintained, but almost no programs are written so someone else can read them.

Don Knuth is changing that. I recently spent a couple of pleasant evenings reading the five-hundred-page implementation of the TEX document compiler. I have no intention of modifying the code, nor am I much more interested in document compilers than the average programmer-on-the-street. I read the code, rather, for the same reason that a student of architecture would spend an afternoon admiring one of Frank Lloyd Wright's buildings. There was a lot to admire in Knuth's work: the decomposition of the large task into subroutines, elegant algorithms and data structures, and a coding style that gives a robust, portable, and maintainable system. I'm a better programmer for having read the program, and I had a lot of fun doing it.

At this point, of course, I hope that you'll run out and read the TEX program yourself; the Further Reading tells you where to find it. As a temporary substitute, this column introduces the programming style that Knuth used to create his program, and the WEB programming system that supports the approach. He calls the style "literate programming"; his goal is to produce programs that are works of literature. My dictionary defines literature as "writings having excellence of form or expression and expressing ideas of permanent or universal interest." I think that Knuth has met his goal.

This column describes the style and presents a small example that Don Knuth was kind enough to write; next month's column is devoted to a more substantial literate program by Knuth.

The Vision

When I wrote my first program, the only reader I had in mind was the computer that ran it. The "structured programming" revolution of the early 1970s taught us that we should keep in mind several other purposes of a program:

> *Design.* As I write a program, I should use a language that minimizes the distance between the problem-solving strategies I have in my head and the program text I eventually write on paper.
>
> *Analysis.* When I develop particularly subtle code, I should use a language that helps me to reason about its correctness.
>
> *Maintenance.* When I write a program, I should keep in mind that its next reader might be someone who is totally unfamiliar with it (such as myself, a year later).

These insights had a tremendous impact. A few principles of programming style and a little discipline led to Cobol, Fortran, and assembly routines that were easier to understand. By the early 1980s, most of us had stopped debating whether `goto` statements were acceptable and had started programming in a high-level language that encouraged cleanliness of expression.

This raised the problem one level: we can understand any given procedure, but it's still hard to make sense of the system as a whole ("I see the trees, but where is the forest?"). Researchers have worked on many kinds of solutions to this problem, such as documentation techniques and module specification and interconnection languages.

Knuth's insight is to focus on the program as a message from its author to its readers. While typical programs are organized for the convenience of their compilers, literate programs are designed for their human readers. At some point, of course, the program must be executed by a computer. Knuth's system allows the programmer to think at a high level, and has the computer do the dirty work of translating the literate description into an executable program.

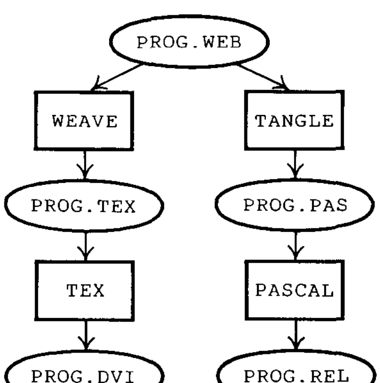
Before we move on to the details of the system, take a few minutes to enjoy Knuth's Program 1 on pages 366–367. In addition to illustrating literate programming, it is also a particularly efficient solution to a problem posed in an earlier column.

The WEB System

O what a tangled web we weave
When first we practice to deceive!
WALTER SCOTT

Program 1 may look too good to be true, but it is indeed the genuine article: when Knuth wrote, tested, and debugged the program, he did so from a listing almost exactly like the one presented here.[1] This section will sketch the mechanics of the WEB system and the programming style it encourages.

The major components of a WEB program named PROG are shown in this figure:



The programmer writes the "source file" PROG.WEB. The WEAVE program transforms that file into the TeX input PROG.TEX, which is in turn fed to the TeX compiler. The output of this process (the process is the left branch in the figure) is the file PROG.DVI, a "device-independent" output file that can be printed on a typesetter or laser printer. Program 1 was produced in this fashion.

The same PROG.WEB file can also serve as input to the TANGLE program, which produces the Pascal file PROG.PAS; the Pascal compiler then transforms that to the executable program PROG.REL. Thus the right-hand branch in the above figure yields running code.

Knuth chose the names carefully. The WEB source file is an intricate structure that describes the program both in text and Pascal code. The WEAVE program spins that into a beautiful document; it unites the parts into a coherent whole that can be readily understood by human readers. The TANGLE program, on the other hand, produces a Pascal program that can be processed by a machine, but it is totally unfit for human consumption. (In the bad old days well-intentioned programmers "patched" binary object code; TANGLE output is as ugly as possible to ensure that programmers deal only with WEB files.)

There isn't space enough in this column to give details on the WEB input file PROG.WEB. Parts of it are pure TeX typesetting commands, and other parts are pure Pascal source text. The vast majority, though, is a straightforward combination of English text and program text and a few simple commands to tell which is which. For more details, consult the Further Reading.

But more important than the mechanics of the WEB system is its philosophy. The system does not force one to write in any particular style. Rather, it provides the ability to present the code and text in the order desired by the programmer/author.

The Challenge

When I first read Knuth's "Literate Programming" paper referenced under Further Reading, I was quite impressed by his approach. When I read the large programs referenced there, I was overwhelmed: for the first time, somebody was proud enough of a substantial piece of code to publish it for public viewing, in a way that is inviting to read. I was so fascinated that I wrote Knuth a letter, asking whether he had any spare programs handy that I might publish as a "Programming Pearl."

But that was too easy for Knuth. He responded, "Why should you let me choose the program? My claim is that programming is an artistic endeavor and that the WEB system gives me the best way to write beautiful programs. Therefore I should be able to meet a stiffer test: I should be able to write a superliterate program that will be noticeably better

[1] Only "almost exactly" because his program was re-typeset to conform to *Communications* style. Knuth produced the program in the right size and shape, but he didn't bother with details such as spacing, font families, and ragged-right text justification. We also deleted the index and the table of contents to squeeze the program onto two pages; next month's program contains both.

PROGRAM 1. A Small Work of Literature by D. E. Knuth

1. Introduction. Jon Bentley recently discussed the following interesting problem as one of his "Programming Pearls" [*Communications of the ACM 27* (December, 1984), 1179–1182]:

> The input consists of two integers M and N, with $M < N$. The output is a sorted list of M random numbers in the range $1 \,..\, N$ in which no integer occurs more than once. For probability buffs, we desire a sorted selection without replacement in which each selection occurs equiprobably.

The present program illustrates what I think is the best solution to this problem, when M is reasonably large yet small compared to N. It's the method described tersely in the answer to exercise 3.4.2–15 of my book *Seminumerical Algorithms*, pp. 141 and 555.

2. For simplicity, all input and output in this program is assumed to be handled at the terminal. The WEB macros *read_terminal*, *print*, and *print_ln* defined here can easily be changed to accommodate other conventions.

define *read_terminal* (#) ≡ *read*(*tty*, #)
  {input a value from the terminal}
define *print* (#) ≡ *write*(*tty*, #)
  {output to the terminal}
define *print_ln* (#) ≡ *write_ln* (*tty*, #)
  {output to the terminal and end the line}

3. Here's an outline of the entire Pascal program:

program *sample*;
  var ⟨Global variables 4⟩
  ⟨The random number generation procedure 5⟩
  begin ⟨The main program 6⟩;
  end.

4. The global variables M and N have already been mentioned; we had better declare them. Other global variables will be declared later.

define *M_max* = 5000 {maximum value of M allowed in this program}

⟨Global variables 4⟩ ≡
M: *integer*; {size of the sample}
N: *integer*; {size of the population}
See also sections 7, 9, and 13.
This code is used in section 3.

5. We assume the existence of a system routine called *rand_int*(i, j) that returns a random integer chosen uniformly in the range $i \,..\, j$.

⟨The random number generation procedure 5⟩ ≡
function *rand_int*(i, j : *integer*): *integer*; *extern*;
This code is used in section 3.

6. A plan of attack. After the user has specified M and N, we compute the sample by following a general procedure recommended by Bentley:

⟨The main program 6⟩ ≡
  ⟨Establish the values of M and N 8⟩;
  size ← 0; ⟨Initialize set S to empty 10⟩;
  while *size* < M **do**
    begin T ← *rand_int*(1, N);
    ⟨If T is not in S, insert it and increase *size* 11⟩;
    end;
  ⟨Print the elements of S in sorted order 14⟩
This code is used in section 3.

7. The main program just sketched has introduced several more globals. There's a set S of integers, whose representation will be deferred until later; but we can declare two auxiliary integer variables now.

⟨Global variables 4⟩ +≡
size: *integer*; {the number of elements in set S}
T: *integer*; {new candidate for membership in S}

8. The first order of business is to have a short dialogue with the user.

⟨Establish the values of M and N 8⟩ ≡
  repeat *print*('population␣size:␣N␣=␣');
    read_terminal(N);
    if $N \le 0$ **then**
      print_ln('N␣should␣be␣positive!');
  until $N > 0$;
  repeat *print*('sample␣size:␣M␣=␣');
    read_terminal(M);
    if $M < 0$ **then**
      print_ln('M␣',
        'shouldn''t␣be␣negative!')
    else if $M > N$ **then**
      print_ln('M␣shouldn''t␣exceed␣N!')
    else if $M > M\_max$ **then**
      print_ln('(Sorry,␣',
        'M␣must␣be␣at␣most␣',
        M_max:1,'.)');
  until $(M \ge 0) \land (M \le N) \land (M \le M\_max)$
This code is used in section 6.

PROGRAM 1. Knuth's Program, Continued

9. An ordered hash table. The key idea to an efficient solution of this sampling problem is to maintain a set whose entries are easily sorted. The method of "ordered hash tables" [Amble and Knuth, *The Computer Journal 17* (May 1974), 135–142] is ideally suited to this task, as we shall see.

Ordered hashing is similar to ordinary linear probing, except that the relative order of keys is taken into account. The cited paper derives theoretical results that will not be rederived here, but we shall use the following fundamental property: *The entries of an ordered hash table are independent of the order in which its keys were inserted.* Thus, an ordered hash table is a "canonical" representation of its set of entries.

We shall represent S by an array of $2M$ integers. Since Pascal doesn't permit arrays of variable size, we must leave room for the largest possible table.

```
⟨Global variables 4⟩ +≡
hash: array [0 . . M_max + M_max − 1] of integer;
        {the ordered hash table}
H: 0 . . M_max + M_max − 1;   {an index into hash}
H_max: 0 . . M_max + M_max − 1;
        {the current hash size}
alpha: real;   {the ratio of table size to N}
```

10.
```
⟨Initialize set S to empty 10⟩ ≡
  H_max ← 2 * M − 1; alpha ← 2 * M/N;
  for H ← 0 to H_max do hash[H] ← 0
```
This code is used in section 6.

11. Now we come to the interesting part, where the algorithm tries to insert T into an ordered hash table. The hash address $H = \lfloor 2M(T-1)/N \rfloor$ is used as a starting point, since this quantity is monotonic in T and almost uniformly distributed in the range $0 \le H < 2M$.

```
⟨If T is not in S, insert it and increase size 11⟩ ≡
  H ← trunc(alpha * (T − 1));
  while hash[H] > T do
    if H = 0 then H ← H_max else H ← H − 1;
  if hash[H] < T then   {T is not present}
    begin size ← size + 1;
    ⟨Insert T into the ordered hash table 12⟩;
    end
```
This code is used in section 6.

12. The heart of ordered hashing is the insertion process. In general, the new key T will be inserted in place of a previous key $T_1 < T$, which is then reinserted in place of $T_2 < T_1$, etc., until an empty slot is discovered.

```
⟨Insert T into the ordered hash table 12⟩ ≡
  while hash[H] > 0 do
    begin TT ← hash[H];   {we have 0 < TT < T}
    hash[H] ← T; T ← TT;
    repeat if H = 0 then H ← H_max
      else H ← H − 1;
    until hash[H] < T;
    end;
  hash[H] ← T
```
This code is used in section 11.

13.
```
⟨Global variables 4⟩ +≡
TT: integer;   {a key that's being moved}
```

14. Sorting in linear time. The climax of this program is the fact that the entries in our ordered hash table can easily be read out in increasing order.

Why is this true? Well, we know that the final state of the table is independent of the order in which the elements entered. Furthermore it's easy to understand what the table looks like when the entries are inserted in decreasing order, because we have used a monotonic hash function. Therefore we know that the table must have an especially simple form.

Suppose the nonzero entries are $T_1 < \cdots < T_M$. If k of these have "wrapped around" in the insertion process (i.e., if H passed from 0 to *H_max*, k times), table position *hash*[0] will either be zero (in which case k must also be zero) or it will contain T_{k+1}. In the latter case, the entries $T_{k+1} < \cdots < T_M$ and $T_1 < \cdots < T_k$ will appear in order from left to right. Thus the output can be sorted with at most two passes over the table!

```
define print_it ≡ print_ln(hash[H] : 10)

⟨Print the elements of S in sorted order 14⟩ ≡
  if hash[0] = 0 then   {there was no wrap-around}
    begin for H ← 1 to H_max do
      if hash[H] > 0 then print_it;
    end
  else begin for H ← 1 to H_max do
          {print the wrapped-around entries}
      if hash[H] > 0 then
        if hash[H] < hash[0] then print_it;
    for H ← 0 to H_max do
      if hash[H] ≥ hash[0] then print_it;
    end
```
This code is used in section 6.

than an ordinary one, whatever the topic. So how about this: You tell me what sort of program you want me to write, and I'll try to prove the merits of literate programming by finding the best possible solution to whatever problem you pose[2]—at least the best by current standards."

He laid some ground rules for the task. The program had to be short enough to fit comfortably in a column, say, an afternoon's worth of programming. It had to be a complete program (not just a fragment), and could not stress input and output (Knuth has boilerplate to handle that problem, but that isn't of interest to most readers). Because his "Literate Programming" article is built around a program to print prime numbers, this assignment should avoid number-theoretic problems.

I chose a problem that I had assigned to several classes on data structures.

> Given a text file and an integer *K*, you are to print the *K* most common words in the file (and the number of their occurrences) in decreasing frequency.

I left open a number of issues, such as the precise definition of words and limits on sizes such as maximum number of words. I did impose an efficiency constraint: a user should be able to find the 100 most frequent words in a twenty-page technical paper without undue emotional trauma.

This problem has several pleasant attributes: it combines simple text manipulation with searching (to increment the count of this input word) and sorting (for output in decreasing frequency). Furthermore, it's useful: I run such a program on documents I write, to find overused words.

Next month's column presents Knuth's literate solution to this problem. Problem 1 encourages you to tackle the problem yourself to increase your appreciation of Knuth's program.

Principles

An Important Problem. Most real programs are written to be executed but not read; many published programs are written to be read but have never been executed.[3] Knuth's work on literate programs is an important step towards programs fit for both man and computing beast. That's good news for writers as well as readers: Rob Pike writes, "Publishing programs is a healthy habit. Every program I've written knowing it was to be published was improved by that knowledge. I think more clearly when I'm writing for an audience, and find it helps to pretend there always is one."

An Important Solution. In addition to defining (and naming) the area, Knuth has made two fundamental contributions to literate programming. The first is his WEB system, which has been used to develop several large (and widely used) programs. His insights, though, transcend the particular system: his "Literate Programming" paper describes WEB look-alikes implemented for other programming and document-production languages. The second fundamental contribution is a body of literate programs written in WEB, several of which are referenced under Further Reading. Most computer scientists are as cowardly as I am; our published programs are rarely more than tiny (and highly polished) subroutines. Knuth is almost unique in publishing the code to workhorse programs. He even believes that it is correct: in the book *TEX: The Program* he writes that "I believe that the final bug in TEX was discovered and removed on November 27, 1985" and offers the princely sum of $20.48 to the finder of any error still lurking in the code.

Problems in Paradise. Because it is based on Pascal, WEB inherits all the universality and some of the problems of the language (although it nicely patches several serious defects of Pascal). A WEB program is written in a mixture of WEB, TEX, and Pascal; that can be a barrier both for learning to use the system and for debugging a program. And the very name "literate programming" implies that its practitioners must be competent in both literature and programming; it is hard enough to find people with one of those skills, let alone both (though WEB does amplify one's abilities).

Problems

1. Knuth's programming problem (finding the *K* most common words in a document) can be interpreted in several ways; Knuth's assignment is somewhere between a and b. Try the problem yourself in one or more of these versions.
   a. *An exercise in simple programming.* In an Algol-like language, implement the simplest program to solve the problem (simplicity might be measured by lines of source code).
   b. *An exercise in efficient programming.* In an Algol-like language, implement the most efficient program to solve the problem (measured in terms of time and/or space).

[2] Although I assigned the program to be described next month, Knuth chose Program 1 himself. When I sent him the "assignment" described above, he returned both the requested solution and a solution to a problem described in an earlier column. He has kindly allowed both programs to be published.

[3] There are exceptions. The programs in Kernighan and Plauger's *Software Tools* are widely used and were included in the text directly from their executable form (the book and the programs were published by Addison-Wesley in 1976; a Pascal version appeared in 1981). I am less exacting with the small programs that appear in this column: I usually test and debug them in a real language (typically C or AWK), then transliterate the trusted code into the Pascal-like pseudocode that I use in the column.

c. *An exercise in text processing.* The February column discussed novel solutions to hard problems. Can you find a way to use existing text processing tools to solve this problem with very little new code?

2. Implement Knuth's Program 1 in your favorite language, using the best documentation style that you know. How does it compare to Program 1 in length and comprehensibility?
3. Analyze the run time taken by Program 1, either mathematically or experimentally.
4. Knuth's Program 1 solves the sampling problem in $O(M)$ expected time and $O(M)$ space; show how it can be solved in $O(M)$ expected time and $O(1)$ space.
5. [H. Trickey] One can view WEB as providing two levels of macros: one can **define** a short string or use the ⟨Do this now⟩ notation for longer pieces of code. Is this mechanism qualitatively better than that provided by other programming environments?
6. [H. Trickey] TANGLE intentionally produces unreadable code. Are there any potential problems?
7. [D. E. Knuth] A program for "set equality" must determine whether two input sequences of integers determine the same set. Show how to use ordered hash tables to solve this problem.

Further Reading

"Literate programming" is the title and the topic of Knuth's article in the May 1984 *Computer Journal* (Volume 27, Number 2, pp. 97–111). It introduces a literate style of programming with the example of printing the first 1000 prime numbers. Complete documentation of "The WEB System of Structured Documentation" is available as Stanford Computer Science technical report 980 (September 1983, 206 pages); it contains the WEB source code for TANGLE and WEAVE.

The small programs in this column and next month's hint at the benefits of literate programming; its full power can only be appreciated when you see it applied to substantial programs. Two large WEB programs appear in Knuth's five-volume *Computers and Typesetting*, just published by Addison-Wesley. The source code for TEX is Volume B, entitled *TEX: The Program* (xvi + 594 pages). Volume D is *METAFONT: The Program* (xvi + 560 pages). Volume A is *The TEXbook*, Volume C is *The METAFONTbook*, and Volume E is *Computer Modern Typefaces*.

For Correspondence: Jon Bentley, AT&T Bell Laboratories, Room 2C-317, 600 Mountain Ave., Murray Hill, NJ 07974.