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 under-
stand 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 have 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:

![WEB System Diagram]

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...”
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, printf, and print_in defined here can easily be changed to accommodate other conventions.

```pascal
define read_terminal (#) = read(tty, #)
    {input a value from the terminal}
define printf (#) = write(tty, #)
    {output to the terminal}
define printf_in (#) = write_in (tty, #)
    {output to the terminal and end the line}
```

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

```pascal
program sample;
    var (Global variables)
        M, N: integer;
        size: integer;
        T: integer;
    begin
        (The main program)
            repeat
                print('population size: uN u=');
                read_terminal(N);
                if N < 0 then
                    printf('N should be positive! ');
            until N > 0;
            repeat
                print('sample size: uM u=');
                read_terminal(M);
                if M < 0 then
                    printf('M shouldn't be negative! ');
                    if M > N then
                        printf('M should be less than N! ');
                else if M > M_max then
                    printf('M must be at most M_max! ');
                    printf_in('M_max: ', M_max: 1, ' ');
            until (M > 0) A (M <= N) A (M <= M_max)
    end.
end.
```

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

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

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.

```pascal
function rand_int(i, j: integer): integer; extern;
```

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:

```pascal
begin
    T ← rand_int(1, N);
    if T is not in S, insert it and increase size
end;
```

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.

```pascal
size: integer; {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.

```pascal
repeat
    print('population size: uN u=');
    read_terminal(N);
    if N < 0 then
        printf('N should be positive! ');
    until N > 0;
    repeat
        print('sample size: uM u=');
        read_terminal(M);
        if M < 0 then
            printf('M shouldn't be negative! ');
            if M > N then
                printf('M should be less than N! ');
        else if M > M_max then
            printf('M must be at most M_max! ');
            printf_in('M_max: ', M_max: 1, ' ');
    until (M > 0) A (M <= N) A (M <= M_max)
```
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.

\[
\begin{align*}
\text{Global variables} \quad 4^\dagger + &+ \\
&\text{hash: array } [0 \ldots M_{\text{max}} + M_{\text{max}} - 1] \text{ of integer} ; \quad \text{(the ordered hash table)} \\
&H: 0 \ldots M_{\text{max}} + M_{\text{max}} - 1; \quad \text{(an index into hash)} \\
&H_{\text{max}} : 0 \ldots M_{\text{max}} + M_{\text{max}} - 1; \quad \text{(the current hash size)} \\
&\alpha : \text{real} ; \quad \text{(the ratio of table size to } N) \\
\end{align*}
\]

10. (Initialize set S to empty)

\[
\begin{align*}
H_{\text{max}} &\leftarrow 2 \cdot M - 1; \quad \alpha \leftarrow 2 \cdot M/N; \\
\text{for } H \leftarrow 0 \text{ to } H_{\text{max}} \text{ do } &\text{hash}[H] \leftarrow 0 \\
\end{align*}
\]

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 \( H = L2M(T - 1)/N \) is used as a starting point, since this quantity is monotonic in T and almost uniformly distributed in the range \( 0 \leq H < 2M \).

\[
\begin{align*}
\text{If } T \text{ is not in } S \text{, insert it and increase size} \quad 11) &\equiv \\
&H \leftarrow \text{trunc}(\alpha \cdot (T - 1)); \\
\text{while } &\text{hash}[H] > T \text{ do} \\
&\quad \text{if } H = 0 \text{ then } H \leftarrow H_{\text{max}} \text{ else } H \leftarrow H - 1; \\
&\quad \text{if } \text{hash}[H] < T \then &\text{[T is not present]} \\
&\quad \begin{align*}
&\text{begin } size \leftarrow size + 1; \\
&\text{[Insert T into the ordered hash table 12]} \\
&\text{end}
\end{align*}
\end{align*}
\]

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 re-inserted in place of \( T_2 < T_1 \), etc., until an empty slot is discovered.

\[
\begin{align*}
\text{Insert } T \text{ into the ordered hash table 12) } &\equiv \\
\text{while } &\text{hash}[H] > 0 \text{ do} \\
&\quad \begin{align*}
&\text{begin } TT \leftarrow \text{hash}[H]; \quad \text{[we have } 0 < TT < T] \\
&\text{hash}[H] \leftarrow T; \quad T \leftarrow TT; \\
&\text{repeat if } H = 0 \then \begin{align*}
&H \leftarrow H_{\text{max}} \\
&\text{else } H \leftarrow H - 1; \\
&\text{until } \text{hash}[H] < T; \\
&\text{end;}
\end{align*} \\
&\begin{align*}
&\text{hash}[H] \leftarrow T \\
&\text{This code is used in section 11.}
\end{align*}
\end{align*}
\]

13. (Global variables 4) +=

\[
\begin{align*}
TT: \text{integer} ; \quad \text{[a key that's being moved]} \\
\end{align*}
\]

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 \( T_1 < \ldots < T_M \). If k of these have "wrapped around" in the insertion process (i.e., if \( H \) passed from 0 to \( H_{\text{max}} \), k times), table position \( \text{hash}[0] \) 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 \( T_{k+1} < \ldots < T_M \), and \( T_1 < \ldots < T_k \) will appear in order from left to right. Thus the output can be sorted with at most two passes over the table.

\[
\begin{align*}
\text{define } \text{print-it } &= \text{print-in}[\text{hash}[H] : 10] \\
\text{(Print the elements of } S \text{ in sorted order) } &\equiv \\
&\begin{align*}
&\text{if } \text{hash}[0] = 0 \text{ then } \quad \text{[there was no wrap-around]} \\
&\quad \begin{align*}
&\text{begin } \text{for } H \leftarrow 1 \text{ to } H_{\text{max}} \text{ do } \text{print-it;}
\end{align*} \\
&\text{end}
\end{align*} \\
&\begin{align*}
&\text{else begin for } H \leftarrow 1 \text{ to } H_{\text{max}} \text{ do} \\
&\quad \text{[print the wrapped-around entries]}
\end{align*} \\
&\begin{align*}
&\quad \text{if } \text{hash}[H] > 0 \text{ then } \quad \text{[print the elements of } S \text{ in sorted order]}
\end{align*} \\
&\begin{align*}
&\quad \text{if } \text{hash}[H] < \text{hash}[0] \text{ then print-it;}
\end{align*} \\
&\begin{align*}
&\quad \begin{align*}
&\text{for } H \leftarrow 0 \text{ to } H_{\text{max}} \text{ do} \\
&\quad \text{if } \text{hash}[H] \geq \text{hash}[0] \text{ then print-it;}
\end{align*}
\end{align*}
\]

This code is used in section 6.
Pragramming Pearls

An Important Problem. Most real programs are written to be read but have never been executed. They are written in a real language (typically C or AWK), then transliterate the trusted code into the Pascal-like pseudocode that I use in the column.

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).
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 $O(1)$ 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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

ACM Algorithms

Collected Algorithms from ACM (CALGO) now includes quarterly issues of complete algorithm listings on microfiche as part of the regular CALGO supplement service.

The ACM Algorithms Distribution Service now offers microfiche containing complete listings of ACM algorithms, and also offers compilations of algorithms on tape as a substitute for tapes containing single algorithms. The fiche and tape compilations are available by quarter and by year. Tape compilations covering five years will also be available.

To subscribe to CALGO, request an order form and a free ACM Publications Catalog from the ACM Subscription Department, Association for Computing Machinery, 11 West 42nd Street, New York, NY 10036. To order from the ACM Algorithms Distributions Service, refer to the order form that appears in every issue of ACM Transactions on Mathematical Software.