THE ELEMENTS OF PROGRAMMING STYLE

SECOND EDITION

Kernighan and Plauger

THE ELEMENTS OF PROGRAMMING STYLE

Second Edition

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey

P. J. Plauger

Yourdon, Inc.
New York, New York

McGRAW-HILL BOOK COMPANY

New York St. Louis San Francisco Auckland Bogotá Düsseldorf
Johannesburg London Madrid Mexico Montreal New Delhi
Panama Paris São Paulo Singapore Sydney Tokyo Toronto

Library of Congress Cataloging in Publication Data

Kernighan, Brian W.
The elements of programming style.

Bibliography: p.
Includes index.
1. Electronic digital computers—Programming.
I. Plauger, P.J., date joint author.
II. Title.
QA76.6.K47 1978 001.6'42 78-3498
ISBN 0-07-034207-5

The Elements of Programming Style

234567890 DODO 78321098

This book was set in Times Roman and Courier 12 by the authors, using a Graphic Systems phototypesetter driven by a PDP-11/70 running under the UNIX operating system.

We are deeply indebted to the following authors and publishers for their kind permission to reproduce excerpts from the following copyrighted material:

PREFACE to the Second Edition

The practice of computer programming has changed since *The Elements of Programming Style* first appeared. Programming style has become a legitimate topic of discussion. After years of producing "write-only code," students, teachers, and computing professionals now recognize the importance of readable programs. There has also been a widespread acceptance of structured programming as a valuable coding discipline, and a growing recognition that program design is an important phase, too often neglected in the past.

We have revised *The Elements of Programming Style* extensively to reflect these changes. The first edition avoided any direct mention of the term "structured programming," to steer well clear of the religious debates then prevalent. Now that the fervor has subsided, we feel comfortable in discussing structured coding techniques that actually work well in practice.

The second edition devotes a whole new chapter to program structure, showing how top-down design can lead to better organized programs. Design issues are discussed throughout the text. We have made considerable use of pseudo-code as a program development tool.

We have also rewritten many of the examples presented in the first edition, to reflect (we hope) a greater understanding of how to program well. There are new examples as well, including several from the first edition which now serve as models of how *not* to do things. New exercises have been added. Finally, we have extended and generalized our rules of good style.

We are once again indebted to the authors and publishers who have graciously given us permission to reprint material from their textbooks. Looking back on some of our own examples makes us realize how demanding an effort good programming is.

We would also like to thank friends who read the second edition in draft form. In particular, Al Aho, Jim Blue, Stu Feldman, Paul Kernighan, Doug McIlroy, Ralph Muha, and Dick Wexelblat provided us with valuable suggestions.

Brian W. Kernighan

P. J. Plauger

PREFACE to the First Edition

Good programming cannot be taught by preaching generalities. The way to learn to program well is by seeing, over and over, how real programs can be improved by the application of a few principles of good practice and a little common sense. Practice in critical reading leads to skill in rewriting, which in turn leads to better writing.

This book is a study of a large number of "real" programs, each of which provides one or more lessons in style. We discuss the shortcomings of each example, rewrite it in a better way, then draw a general rule from the specific case. The approach is pragmatic and down-to-earth; we are more interested in improving current programming practice than in setting up an elaborate theory of how programming should be done. Consequently, this book can be used as a supplement in a programming course at any level, or as a refresher for experienced programmers.

The examples we give are all in Fortran and PL/I, since these languages are widely used and are sufficiently similar that a reading knowledge of one means that the other can also be *read* well enough. (We avoid complicated constructions in either language and explain unavoidable idioms as we encounter them.) *The principles of style, however, are applicable in all languages, including assembly codes.*

Our aim is to teach the elements of good style in a small space, so we concentrate on essentials. Rules are laid down throughout the text to emphasize the lessons learned. Each chapter ends with a summary and a set of "points to ponder," which provide exercises and a chance to investigate topics not fully covered in the text itself. Finally we collect our rules in one place for handy reference.

A word on the sources of the examples: *all* of the programs we use are taken from programming textbooks. Thus, we do not set up artificial programs to illustrate our points — we use finished products, written and published by experienced programmers. Since these examples are typically the first code seen by a novice programmer, we would hope that they would be models of good style. Unfortunately, we sometimes find that the opposite is true — textbook examples often demonstrate the state of the art of computer programming all too well. (We have done our best to play fair — we don't think that any of the programs are made to look bad by being quoted out of context.)

Let us state clearly, however, that we intend no criticism of textbook authors, either individually or as a class. Shortcomings show only that we are all human, and that under the pressure of a large, intellectually demanding task like writing a program or a book, it is much too easy to do some things imperfectly. We have no

doubt that a few of our "good" programs will provide "bad" examples for some future writer — we hope only that he and his readers will learn from the experience of studying them carefully.

A manual of programming style could not have been written without the pioneering work of numerous people, many of whom have written excellent programming textbooks. D. D. McCracken and G. M. Weinberg, for instance, have long taught the virtues of simplicity and clarity. And the work of E. W. Dijkstra and Harlan Mills on structured programming has made possible our rules for properly specifying flow of control. The form and approach of this book has been strongly influenced by *The Elements of Style* by W. Strunk and E. B. White. We have tried to emulate their brevity by concentrating on the essential practical aspects of style.

We are indebted to many people for their help and encouragement. We would like especially to thank the authors and publishers who gave us permission to reproduce the computer programs used in this text. Their cooperation is greatly appreciated.

Our friends and colleagues at Bell Laboratories provided numerous useful suggestions, which we have incorporated, and saved us from more than one embarrassing blunder, which we have deleted. In particular, V. A. Vyssotsky bore with us through several revisions; for his perceptive comments and enthusiastic support at every stage of this book's evolution (and for several aphorisms we have shamelessly stolen) we are deeply grateful. We would also like to single out A. V. Aho, M. E. Lesk, M. D. McIlroy, and J. S. Thompson for the extensive time and assistance they gave to this project.

We were able to type the manuscript directly into a PDP 11/45, edit the source, check the programs, and set the final version in type — all with the help of a uniquely flexible operating system called UNIX. K. L. Thompson and D. M. Ritchie were the principal architects of UNIX; besides reading drafts, they helped us get the most out of the system while we were working on this book. J. F. Ossanna wrote the typesetting program and made several modifications for our special needs. We thank them.

Brian W. Kernighan

P. J. Plauger

THE ELEMENTS OF PROGRAMMING STYLE

CONTENTS

CHAPTER 1: INTRODUCTION

Consider the program fragment

```
      DO 14 I=1,N
      DO 14 J=1,N
   14 V(I,J)=(I/J)*(J/I)
```

A modest familiarity with Fortran tells us that this doubly nested DO loop assigns something to each element of an N by N matrix V. What are the values assigned? I and J are positive integer variables and, in Fortran, integer division truncates toward zero. Thus when I is less than J, (I/J) is zero; conversely, when J is less than I, (J/I) is zero. When I equals J, both factors are one. So (I/J)*(J/I) is one if and only if I equals J; otherwise it is zero. The program fragment puts ones on the diagonal of V and zeros everywhere else. (V becomes an identity matrix.) How clever!

Or is it?

Suppose you encountered this fragment in a larger program. If your knowledge of Fortran is sufficiently deep, you may have enjoyed the clever use of integer division. Possibly you were appalled that two divisions, a multiplication, and a conversion from integer to floating point were invoked when simpler mechanisms are available. More likely, you were driven to duplicating the reasoning we gave above to understand what is happening. Far more likely, you formed a vague notion that something useful is being put into an array and simply moved on. Only if motivated strongly, perhaps by the need to debug or to alter the program, would you be likely to go back and puzzle out the precise meaning.

A better version of the fragment is

```
C  MAKE V AN IDENTITY MATRIX
      DO 14 I = 1,N
         DO 12 J = 1,N
   12       V(I,J) = 0.0
   14    V(I,I) = 1.0
```

This zeros each row, then changes its diagonal element to one. The intent is now reasonably clear, and the code even happens to execute faster. Had we been programming in PL/I, we could have been more explicit:

```
/* MAKE V AN IDENTITY MATRIX */
      V = 0.0;
      DO I = 1 TO N;
         V(I,I) = 1.0;
      END;
```

In either case, it is more important to make the purpose of the code unmistakable than to display virtuosity. Even storage requirements and execution time are unimportant by comparison, for setting up an identity matrix must surely be but a small part of the whole program. The problem with obscure code is that debugging and modification become much more difficult, and these are already the hardest aspects of computer programming. Besides, there is the added danger that a too-clever program may not say what you thought it said.

Write clearly — don't be too clever.

Let's pause for a moment and look at what we've done. We studied part of a program, taken verbatim from a programming textbook, and discussed what was good about it and what was bad. Then we made it better. (Not necessarily perfect — just better.) And then we drew a rule or a general conclusion from our analysis and improvements, a rule that would have sounded like a sweeping generality in the abstract, but which makes sense and can be applied once you've seen a specific case.

The rest of the book will be much the same thing — an example from a text, discussion, improvements, and a rule, repeated over and over. When you have finished reading the book, you should be able to criticize your own code. More important, you should be able to write it better in the first place, with less need for criticism.

We have tried to sort the examples into a logical progression, but as you shall see, real programs are like prose — they often violate simultaneously a number of rules of good practice. Thus our classification scheme may sometimes seem arbitrary and we will often have to digress.

Most of the examples will be bigger than the one we just saw, but not excessively so; with the help of our discussion, you should be able to follow them even if you're a beginner. In fact, most of the bigger programs will shrink before your very eyes as we modify them. Sheer size is often an illusion, reflecting only a need for improvement.

The examples are all in either Fortran or PL/I, but if one or both of these languages is unfamiliar, that shouldn't intimidate you any more than size should. Although you may not be able to write a PL/I program, say, you will certainly be able to read one well enough to understand the point we are making, and the practice in reading will make learning PL/I that much easier.

For example, here is a small part of a PL/I program that we will discuss in detail in Chapter 4:

```
            IF CTR > 45 THEN GO TO OVFLO;
          ELSE GO TO RDCARD;
OVFLO:
      ...
```

The first GOTO simply goes around the second GOTO, which seems a bit disorganized. If we replace > by <=, we can write

```
      IF CTR <= 45 THEN GOTO RDCARD;
OVFLO:
      ...
```

One less statement, simpler logic, and, as it happens, we no longer need the label OVFLO. The lesson? Don't branch around branches: turn relational tests around if it makes the program easier to understand. We will soon see a Fortran example of exactly the same failing, which brings up an important point: although details vary from language to language, *the principles of style are the same.* Branching around branches is confusing in any language. So even though you program in Cobol or Basic or assembly language or whatever, the guidelines you find here still apply.

It might seem that we're making a great fuss about a little thing in this last example. After all, it's still pretty obvious what the code says. The trouble is, although any single weakness causes no great harm, the cumulative effect of several confusing statements is code that is simply unintelligible.

Our next example is somewhat larger:

The following is a typical program to evaluate the square root (B) *of a number* (X):

```
      READ(5,1)X
    1 FORMAT(F10.5)
      A=X/2
    2 B=(X/A+A)/2
      C=B-A
      IF(C.LT.0)C=-C
      IF(C.LT.10.E-6)GOTO 3
      A=B
      GOTO 2
    3 WRITE(6,1)B
      STOP
      END
```

Because it is bigger, we can study it on several levels and learn something from each. For instance, before we analyze the code in detail, we might consider whether this program is truly "typical." It is unlikely that a square root routine would be packaged as a main program that reads its input from a file — a function with an argument would be far more useful. Even assuming that we really do want a main program that computes square roots, is it likely that we would want it to compute only one before stopping?

This unfortunate tendency to write overly restricted code influences how we write programs that are supposed to be general. Soon enough we shall meet programs designed to keep track of exactly seventeen salesmen, to sort precisely 500 numbers, to trace through just one maze. We can only guess at how much of the program rewriting that goes on every day actually amounts to entering parameters via the compiler.

Let us continue with the square root program. It is an implementation of Newton's method, which is indeed at the heart of many a library square root routine (although we need not go into precisely how it works). With proper data, the method converges rapidly. If X is negative, however, this program can go into an infinite loop. (Try it.) A good routine would instead provide an error return or a diagnostic message. And the program blows up in statement 2 if X is zero, a case that must be treated separately. The square root of zero should be reported as zero.

Even for strictly positive values of X this program can give garbage for an answer. The problem lies in the convergence test used:

```
C=B-A
IF(C.LT.0)C=-C
IF(C.LT.10.E-6)GOTO 3
```

To make effective use of the Fortran language, the second line should read

```
C = ABS(C)
```

To avoid having someone misread 10.E-6 as "10 to the minus sixth power," the constant in the third line should be 1.0E-5 or even 0.00001. And to say what is meant without bombast, all three lines should be changed to

```
IF (ABS(B-A) .LT. 1.0E-5) GOTO 3
```

The test now reads clearly; it is merely wrong.

If X is large, it is quite possible that the absolute difference between successive trial roots will never be less than the arbitrary threshold of 1.0E-5 unless it is exactly zero, because of the finite precision with which computers represent numbers. It is a delicate question of numerical analysis whether this difference will always become zero. For small values of X, on the other hand, the criterion will be met long before a good approximation is attained. But if we replace the absolute convergence criterion by a test of whether the estimate is close enough *relative to the original data,* we should get five place accuracy for most positive arguments:

```
C COMPUTE SQUARE ROOTS BY NEWTON'S METHOD
 100  READ(5,110) X
 110      FORMAT(F10.0)
C
      IF (X .LT. 0.0) WRITE(6,120) X
 120      FORMAT(1X, 'SQRT(', 1PE12.4, ') UNDEFINED')
C
      IF (X .EQ. 0.0) WRITE(6,130) X, X
 130      FORMAT(1X, 'SQRT(', 1PE12.4, ') = ', 1PE12.4)
C
      IF (X .LE. 0.0) GOTO 100
      B = X/2.0
 200  IF (ABS(X/B - B) .LT. 1.0E-5 * B) GOTO 300
      B = (X/B + B) / 2.0
      GOTO 200
 300  WRITE(6,130) X, B
      GOTO 100
      END
```

The modified program is still not a typical square root routine, nor do we wish to go into the detailed treatment of floating point arithmetic needed to make it one. The original example is, however, typical of programs in general: it profits from

criticism and revision.

Let us conclude the chapter with another example that illustrates several failings. This program is a sorting routine.

```
      DIMENSION N(500)
      WRITE (6,6)
    6 FORMAT (1H1,26HNUMBERS IN ALGEBRAIC ORDER)
      DO 8 I=1,500
    8 READ (5,7) N(I)
    7 FORMAT (I4)
      DO 10 K=1,1999
      J=K-1000
      DO 10 I-1,500
      IF(N(I)-J)10,9,10
   10 CONTINUE
      STOP
    9 WRITE (6,95) N(I)
   95 FORMAT (1H ,I4)
      GO TO 10
      END
```

The code suffers not only from lack of generality, but from an ill-advised algorithm, some dubious coding practices, and even a typographical error. The line

```
      DO 10 I-1,500
```

is wrong: the "-" should be "=". The program was contrived in part to illustrate that the range of a DO loop can be extended by a transfer outside and back, even though in this case the inner DO loop *and* the code of the extended range can all be better written in line as

```
      DO 10 I = 1, 500
         IF (N(I) .EQ. J) WRITE (6,95) N(I)
   95       FORMAT(1X, I4)
   10 CONTINUE
```

More to the point is the question of whether programmers should be encouraged to use extended ranges in the first place. Jumping around unnecessarily in a computer program has proved to be a fruitful source of errors, and usually indicates that the programmer is not entirely in control of the code. The apparently random statement numbers in this example are often a symptom of the same disorder.

The program has other flaws. It reads in 500 numbers, one per card, and sorts them about as inefficiently as possible — by comparing each number with all integers between −999 and +999. It does this once, for only one set of numbers, then stops.

But wait. With an I4 input format, it is possible to read positive numbers as large as 9999, since we can leave out the plus sign; the program as it stands will fail to list four-digit numbers. To correct the oversight will slow the algorithm by a factor of more than five, without extending its generality in the least. Extending this method to handle larger integers would slow it by orders of magnitude, and to ask it to handle floating point numbers would be unthinkable.

We will not attempt to rewrite this code, since we disagree with its basic approach. (Chapter 7 contains several better sorting programs.) We just want to

show that the same program can be viewed from different perspectives, and that the job of critical reading doesn't end when you find a typo or even a poor coding practice. In the chapters to come we will explore the issues touched on here and several others that strongly affect programming style.

We begin, in Chapter 2, with a study of how to express individual statements clearly. Writing arithmetic expressions and conditional (`IF`) statements is usually the first aspect of computer programming that is taught. It is important to master these fundamentals before becoming too involved with other language features.

Chapter 3 treats the control-flow structure of computer programs, that is, how flow of control is specified through looping and decision-making statements. It also shows how data can be represented to make programming as easy as possible, and how data structure can be used to derive a clean control flow. Program structure is covered in Chapter 4, how to break up a program into manageable pieces. Considerable emphasis is given in these chapters to proper use of structured programming and sound design techniques.

Chapter 5 examines input and output: how to render programs less vulnerable to bad input data and what to output to obtain maximum benefit from a run. A number of common blunders are studied in Chapter 6, and tips are given on how to spot such errors and correct them.

Contrary to popular practice, efficiency and documentation are reserved for the last two chapters, 7 and 8. While both of these topics are important and warrant study, we feel they have received proportionately too much attention — particularly in introductory courses — at the expense of clarity and general good style.

A few words on the ground rules we have used in criticizing programs:

(1) Programs are presented in a form as close to the original as our typescript permits. Formatting, typographical errors, and syntax errors are as in the original. (Exception: three PL/I programs have been translated from the 48-character set into the 60-character set.)

(2) We regularly abstract parts of programs to focus better on the essential points. We believe that the failings we discuss are inherent in the code shown, and not caused or aggravated by abstracting. We have tried not to quote out of context. We have tried throughout to solve essentially the same problem as the original version did, so comparisons may be made fairly, even though this sometimes means that we do not make all possible improvements in programs.

(3) We will not fault an example for using non-standard language features (for example, mixed mode arithmetic in Fortran) unless the use is quite unusual or dangerous. Most compilers accept non-standard constructions, and standards themselves change with time. Remember, though, that unusual features are rarely portable, and are the least resistant to changes in their environment.

Our own Fortran hews closely to the 1966 American National Standards Institute (ANSI) version, except for our use of quoted Hollerith strings (we refuse to count characters). PL/I programs meet the standard set by IBM's checkout compiler, version 1, release 3.0. Although there are new versions of Fortran and PL/I in sight which will make better programming possible in both of these

languages, they are not yet widespread, so we have not written any examples in the newer dialects.

(4) In our discussions of numerical algorithms (like the square root routine above) we will not try to treat all possible pathological cases; the defenses needed against overflow, significance loss, and other numerical pitfalls are beyond the scope of this book. But we do insist that at least the rudimentary precautions be taken, like using relative tests instead of absolute and avoiding division by zero, to ensure good results for reasonable inputs.

(5) Every line of code in this book has been compiled, directly from the text, which is in machine-readable form. All of our programs have been tested (Fortran on a Honeywell 6070, PL/I on an IBM 370/168). Our Fortran programs have also been run through a verifier to monitor compliance with the ANSI standard.

Nevertheless, mistakes can occur. We encourage you to view with suspicion anything we say that looks peculiar. Test it, try it out. Don't treat computer output as gospel. If you learn to be wary of everyone else's programs, you will be better able to check your own.