Programming in style

Since this book is about using a computer to accomplish useful work in the numeric-intensive areas of science, engineering and modern finance, the reader might be tempted to skip a chapter dealing primarily with issues of programming style. In my graduate student days I would have been so tempted myself. Sad experience has taught me the value of doing things right the first time, even though it means taking some extra time and trouble.

To make the presentation as painless as possible I have compiled some programming precepts that—if followed conscientiously—will lead to programs that work correctly and can be exhibited proudly to one's peers. These ideas apply to any programming language, from pure assembly code to the most abstract or arcane. I will illustrate with examples from Fortran, C and Forth.

1. Elementary considerations

Although personal computers have gotten so fast and cheap[1] that we rarely worry about program efficiency, anyone crunching numbers in a scientific or engineering context must worry to some extent about speed. Of course if the program gives wrong answers it does not matter how fast it produced them, so we will discuss matters pertaining to program correctness in Section below.

The main key to speed is using the right algorithm. Some algorithms are much faster than others. For example sorting a random list can be performed in $O(n \log n)$ time using Quicksort, Heapsort or Mergesort. For a long list this can be orders of magnitude faster than $O(n^2)$ algorithms like bubblesort or insertion sort. Therefore a little study of the literature before writing the program can save a great deal of time in the long run.

An example of the difference between bad and good algorithms is the evaluation of a polynomial:

```
' BASIC subroutine to evaluate a polynomial with real coefficients A(I)
FUNCTION POLY(X, A, N)
DIM A(N)
SUM = 0.
FOR I = 0 TO N
   SUM = SUM + A(I)*X^I
NEXT
POLY = SUM
RETURN
END
```

1. …for example, I own an (obsolete) HP200LX palmtop computer that runs 10 to 100 times faster, and has 20 times the memory, of the multimillion dollar, room-sized IBM 7094 mainframe I used for my PhD thesis work. My desktop is thousands of times faster.

This is a poor routine for a number of reasons. First, to evaluate `X^I` (`X` raised to the `I`'th power) requires—depending on the compiler—either a logarithm, a multiplication and an exponentiation (that is, two transcendental functions and a multiplication); or else `I-1` floating point multiplications. Thus even for a compiler that treats raising to integer powers as a special case, evaluating an Nth order polynomial takes about $N^2/2$ multiplications. There is a faster way to raise numbers to integer powers, that requires about $\log_2 (N)$ multiplications, so the polynomial evaluation time, if the compiler is smart enough to use the fast power method is $O(N \cdot \log_2 N)$. However, Horner's method requires $O(N)$ multiplications, is only a slight modification of the inefficient algorithm, and should therefore be used as a matter of course:

```
' BASIC subroutine to evaluate a polynomial with real coefficients A(I)
' using Horner's method
FUNCTION POLY(X, A, N)
DIM A(N)
SUM = A(N) * X
FOR I = N-1 TO 0 STEP -1
    SUM = (SUM + A(I)) * X
NEXT
POLY = SUM
RETURN
END
```

Other instances where good algorithms make a huge difference in execution speed are sorting and searching, where the wrong algorithm can be $O(N^2)$ and the right one $O(N)$ or $O(N \log_2 N)$.

2. Readability

Readability, Testability, Safety and Maintainability are the attributes of a good computer program. Of course, it should also be correct!—but we are here comparing programs that are algorithmically correct, yet which exhibit the above qualities to varying degrees.

First consider readability. Every language[2] permits the construction of "write-only" code, that once written can be deciphered only with enormous effort. In the worst cases, neither the code's own author nor his personal deity can disencrypt the work after the heat of composition has passed. Hallmarks of write-only code are

- missing or inadequate documentation and commenting;
- cryptic, too-terse, undescriptive names for data structures and functions;

2. Forth is commonly accused of this sin more often than other languages; and indeed some Forth legacy code deserves the epithet; however I have seen plenty of Lisp, C, Awk and Perl codes that are at least as terrifying. I will illustrate with some bad Fortran and Forth.

- convoluted program organization—deeply nested, incorrectly nested or poorly chosen control structures;
- bad factorization (more about this later);
- prolix (*i.e.*, too long) subroutines;
- inappropriate data structures.

Documenting and commenting

When one is intent on solving a computational problem and really only cares about the answer, it is easy to convince oneself that documenting and commenting are of secondary, or even tertiary importance. In such circumstances one thinks, "This is just throwaway code that I'll never look at again, so…" Sad experience teaches that it is exactly this morsel of "throwaway" code—and the accompanying computer output—that one will have to consult in six months or a year because something has changed, because someone has questioned the results, or because we need it for the basis of something new.

To see this problem in action, consider the (uncommented) Fortran function

```
   INTEGER FUNCTION STIB(N,K)
   STIB=0
   DO 10 I=1,K
      NP=N/2
      STIB=N+2*(STIB-NP)
      N=NP
10 CONTINUE
   RETURN
   END
```

What does `STIB` do? Evidently it is a function—that is, for given (implicit integer) arguments it returns another integer computed by some laborious process of dividing and multiplying by 2, performed `K` times. Only the author himself is likely to realize that the subroutine's name, "bits" spelled backwards, suggests that the function reverses the order of bits in the binary representation of an integer lying in the range $\left[0, 2^K - 1\right]$, a process employed in the fast Fourier transform algorithm. For someone who needs the program of which this function is a part, the absence of explanatory comment might entail frustrating hours of deciphering before he gets the point. How much better had the author taken the time to add comments, leading to

```
   INTEGER FUNCTION STIB(N,K)
C
C  Return the bit-reversed integer corresponding to the
C  K least significant (rightmost) bits of N.
C  Algorithm:
C  Shift STIB one bit to the left and add the LSB of N.
C  Shift N one bit to the right. Repeat K times.
C
C                          initialize result
   STIB=0
C                          repeat K times
```

```
   DO 10 I=1,K
C                          right-shift 1 bit
      NP = N/2
C                          left-shift old result, add LSB of N
      STIB=N+2*(STIB-NP)
C                          replace N by right-shifted version
      N=NP
10 CONTINUE
   RETURN
   END
```

Languages that include in-line comments and bitwise logical operators[3]—BASIC, C, Forth, *e.g.*—permit a clearer statement of the algorithm. For example in C,

```
int stib( int n, int k )   // reverse order of bits of # in [0,2^k-1]
{   int i;
    int j=0;               // initialize answer
    for( i=1; i<=k; i++ )  // loop k times
    {   j=2*j+(n && 1);    // left-shift result and add 1's bit of n
        n=n/2;             // shift n one place to right
    }
    return j;
}
```

For completeness I include the Forth version:

```
: stib  ( n w -- n')          \ reverse order of bits of n in [0,2^w-1]
    LOCALS| w n |             \ def local variable names
    0    ( -- 0)              \ n' = 0 is top of stack
    w 0 DO  ( n')             \ loop w times
            2*                \ left-shift n'
            n 1 AND           \ get 1's bit of n
            +                 \ add to n'
            n  2/  TO n       \ right-shift n -- equiv to n = n/2
        LOOP                  \ end loop
;
```

Documentation in Forth takes two forms. Since Forth uses the machine stack directly for communication between subroutines, we customarily include a "stack comment". The parenthesized comment `( n w -- n')` following the name says that the subroutine `STIB` expects to consume two integers from the stack and leave one (integer) result on the stack. (Despite how it looks, it is a comment, not an argument list.)

The parenthesized comment `( n')` following `DO` is something like an "assertion"[4]—it states that each time the loop is executed it begins with an integer on the stack (and therefore must leave one as a result!).

3. For all I know, later versions of Fortran may provide bitwise operations.

Finally, Forth has the "drop line" comment set off by \ ("back-slash, blank") equivalent to the tick (') in BASIC and the // in C.

Comparing the programs, we see that Fortran's lack of such bitwise logical operators as && or AND made it necessary to obtain the 1's bit of the input by the circuitous route of first setting that bit to zero (right-shift by one bit to make the 1's bit "fall off", then left-shift one bit to replace it with 0); then subtracting the result from the original input. For efficiency we combined the shift operations (synthesized as integer divide- and multiply-by-two) so as to minimize the net number of multiplications and divisions. But this tends to make the algorithm seem rather different from what we were actually trying to do, lowering readability.

Expressing the function in several languages[5] emphasizes the following points:

- omitting comments obscures the algorithm—at the very least every function should be accompanied by a brief statement of its purpose;
- the operators and notational conventions specific to each language account for much of the difference in code readability between languages—that is, when one examines the history of a high level computer language one finds its author(s) concerned with handling more gracefully some specific problem, most often stylistic in nature.

However, comments can be unhelpful. As a case in point I present three versions of the same subroutine for converting to binary a 2-digit decimal number in packed BCD representation[6]. These formulations represent a discussion by three regular contributors to the newsgroup **comp.lang.forth** . (I have taken the liberty of editing their remarks for spelling and clarity.) The first asked

Folks, do you find such style for one-liners readable?

```
: bcd>bin ( uc -- uc' )
\          tens    div_16_mult_10          ones    add
   DUP    0F0 AND   2/ DUP 2/ 2/ +   SWAP   00F AND   +
;
```

The second programmer replied

Your comment indicates bad factoring, especially the one above `2/ DUP 2/ 2/ +`*. It's even wrong, because the result is as if first multiplying by 16, and then dividing by 10 (someone might "optimize" the* `0F0 AND` *out, if he takes the comment literally, and by doing so introduces a bug). Therefore:*

---

4. Assertions are an idea pioneered by C.A.H. Hoare, as a discipline for correct programming. See, *e.g.*, Jon Bentley, *Programming Pearls* (Addison-Wesley, Reading, MA,1989), p. 42 *et seq*. It is as easy to add assertion capability to Forth as to C code.
5. The BASIC version would look very like the Fortran one, except it would have permitted us to say `STIB = 2*STIB + (N AND 1) : N + N\2` and thus shorten the program slightly.
6. Packed BCD stores one (decimal) digit per nybble (4 bits).

```
: *10/16 ( n1 -- n2 )  2/ DUP 2/ 2/ + ; \ much faster than 10 16 */
: tens ( bcdxy -- bcdx0 )  $F0 AND ;
: ones ( bcdxy -- bcd0y )  $0F AND ;
: bcd>bin ( bcd -- n )  DUP tens *10/16 SWAP ones  +  ;
```

You might factor `tens` and `ones` out (as above), but it's not as obscure as the shift-add replacement of division and multiplication. However, if you do more BCD handling, `tens` and `ones` might be good factors, anyway. They improve readability much compared to `<magic number>` AND.

I think this example shows what I mean when I claim commenting often indicates bad factoring, and good word names can replace comments. Factoring in Forth is not just a means to reduce program size!

So if you find yourself commenting some code sequence with a single word (as above), you'd do better to factor the code sequence and name it precisely that word.

The third author also replied to the first:

Yes, it's clear, but I don't like one-liners - can't see the point. For what it's worth this is what I'd do:

```
: BCD>BIN   ( [xy] -- n)          \ two digit packed BCD to binary
    DUP  $F0 and  ( [xy] [x0])    \ get more significant nibble
    2/ dup 2/ 2/ +                \ 1/2 + 1/8 = 10/16,  ms nibble *10/16
    SWAP $0F AND  ( 10*x y)       \ get less significant nibble
    +                             \ n = 10x + y
;
```

The `2/ DUP 2/ 2/ +` is a neat trick - I'll remember that.

What we have seen is that the original clever code—a "hack"[7]—had a descriptive comment over it, but nevertheless interrupted the semantic flow by making the reader stop to think "Huh? Whazzat?". The second (improved) version helped the clarity by giving each operation a descriptive name, so that the final subroutine, `bcd>bin`, simply reads as a description of the algorithm: duplicate the number, get the tens digit and multiply it 10; then get the ones digit and add. The commenting is also more helpful, since he explains why the hack is used rather than the much clearer code that has the same effect: the hack is much faster.

But the third author's approach is the clearest: he not only puts each major step on its own line, he also explains *how* the hack works!

7. We discuss excessively clever tricks and their effect on clarity below.

Control structures

The next aspect of readability has to do with control structures. Programs would be either clumsy or impotent without the ability to execute conditional branches. Based on some condition—a certain number is zero, a certain bit is set—a conditional branch either does nothing (so execution proceeds to the next instruction) or resets the pointer to the next instruction so it points somewhere else in the program. All the standard high-level control structures for looping—`DO`, `FOR`, `WHILE...`—or branching—`IF`, `ELSE`, `CASE...`—are synthesized from one or two conditional branch primitives[8].

As an example of hard-to-read code, consider the Forth subroutine `>FLOAT` for converting a string to a floating point number, shown on the following three pages. This is a required subroutine for ANS Forth systems that implement the FLOATING point wordset. According to the dpANS94 specification,

> "An attempt is made to convert the string specified by c-addr and u to internal floating-point representation. If the string represents a valid floating-point number in the syntax below, its value r and TRUE are returned. If the string does not represent a valid floating-point number only FALSE is returned."

As given, the subroutine exemplifies the besetting sins of sloppy programming style[9]: excessive length, deeply nested control structures, repeated code fragments that should be named and factored into separate subroutines.

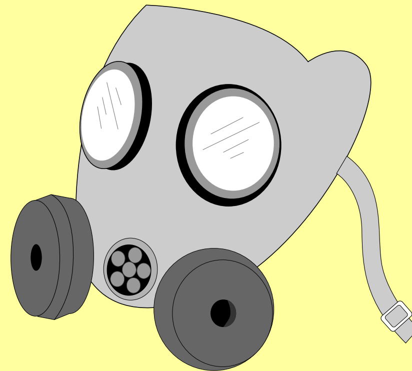
The programmer evidently thought—as so many do—that using a program editor with "pretty-printing" capability (that is, laying out the code with indented control structures, and with the beginnings and ends of control structures vertically aligned) was enough to ensure readability. Well, *ipso facto*, it wasn't.

Once we understand what is going on, the algorithm becomes absurdly simple. In pseudocode it is

```
\ pseudocode algorithm for >FLOAT, assuming IEEE representation
: >FLOAT  ( c-addr u -- flag) ( f: -- r | <nothing>)
   is it properly formed?  IF
      save algebraic sign
      get mantissa \ normalize mantissa ( move decimal point to the left end)
                   \ convert normalized mantissa to fp number
      get exponent \ compute appropriate power of 10 to multiply by
      raise 10 to that power ( exponent + #digits to left of dp)
      multiply  ( F*)
      algebraic sign was negative? IF  FNEGATE  THEN
      leave TRUE flag
   ELSE   leave  FALSE  flag    THEN
;
```

---

8. A primitive is a simple code fragment—it might be at the level of the actual machine code—underlying a user-level instruction. In Forth, *e.g.*, the basic branching primitive is `0BRANCH` which jumps to the (relative) address contained in the memory cell immediately following itself, if the number on top of the stack is 0.
9. And lest the reader think only Forth programmers write such monstrosities, *Numerical Recipes*—in Fortran, C or Pascal—is chock-full of similarly obscure code.



```
: >FLOAT ( c-addr u -- f ) ( f: -- r | <nothing> )
   dup 0=
   IF drop drop false EXIT
   THEN
   128 umin charcnt !    \ save character count
   init->float
   next-char drop     \ get first character
   dup bl =
   IF drop charcnt @ bl skip charcnt !
      next-char
      IF drop drop f0.0
         true EXIT  \ special case of 0.0
      THEN
   THEN
   xsign
   IF    IF 8 ELSE 0 THEN
         19 bcd-char!
         next-char
         IF drop drop false EXIT  THEN
   THEN
   BEGIN digit0    \ check for leading 0's
   WHILE drop   next-char
      IF drop drop f0.0 true EXIT  THEN
      true zerochar !
   REPEAT
   10digit
   IF
      BEGIN $fsignif intcnt @ + c!
         1 intcnt +!
         next-char
         IF drop drop >float-int true EXIT  THEN
      10digit 0=
      UNTIL
      false zerochar !
   THEN
   dup [CHAR] . =    \ decimal point?
   IF drop next-char
      IF drop drop >float-int true EXIT  THEN
      intcnt @ 0=
      IF
         BEGIN digit0
         WHILE drop  next-char
               IF   drop drop f0.0 true EXIT
               ELSE 1 fracnt +!
               THEN
               true zerochar !
         REPEAT
      THEN
      10digit
```

```
    IF false zerochar !
       BEGIN $fsignif intcnt @ + c!
          1 intcnt +!
          1 fracnt +!
          next-char
          IF  drop  drop  >float-int.frac  true EXIT  THEN
          10digit 0=
       UNTIL
    THEN
THEN
e-char
IF     \ exponent indicator
       \  zerochar @   \ optimization, mantissa=0
       \  IF drop f0.0 true EXIT \ then whole number is zero
       \  THEN    \ unfortunately skips validation
       intcnt @ 0=
       IF ( SMuB 07-20-95 drop ) f1.0 19 bcd-char@
          [CHAR] 0 <> ( <-- SMuB 07-20-95 )
          IF fnegate  THEN
       ELSE   >float-int
       THEN
       next-char
       IF drop drop fracnt @
          IF  f10.0  fracnt  @  negate  f**n  f*  THEN
          zerochar @   \ mantissa=0?
          IF fdrop f0.0  THEN        \ then make result 0.0
          true EXIT
       THEN
       xsign
       IF >r next-char
          IF  r>  drop drop drop
              fracnt @
              IF f10.0 fracnt @
                 negate f**n f*
              THEN
              zerochar @  \ mantissa=0?
              IF fdrop f0.0   THEN       \ then make result 0.0

              true EXIT
          THEN
       ELSE 0 >r
       THEN
ELSE   >float-int
       xsign
       IF  >r next-char
          IF r>drop drop drop fdrop false EXIT
                \ SMuB r>drop drop ( SMuB 07-20-95 --> ) drop
                \ SMuB fracnt @
                \ SMuB IF f10.0 fracnt @
                \ SMuB negate f**n f*
                \ SMuB THEN
                \ SMuB true EXIT
          THEN
       ELSE drop drop fdrop false EXIT
       THEN
```

```
THEN
drop 1- charcnt @ 1+ number?
double? 0= and   \ October 1st, 1996 - 10:51 tjz & am
                 \ double exponent not allowed
IF d>s r>
   IF negate  THEN
   fracnt @ negate + f10.0 f**n f*
   zerochar @    \ mantissa=0?
   IF fdrop f0.0  THEN        \ then make result 0.0
   true
ELSE
   2drop fdrop false r>drop EXIT
THEN ;
```

The actual subroutine consists of seven basic operations. By appropriately defining and naming these operations the algorithm becomes almost self-explanatory and needs few additional comments. That is, one might write

```
\ convert a string to a fp number on the fp stack

: fp#?    ( end beg -- flag)  \ test a string for proper format
   skip_sign  skip_digits  skip_dp  skip_digits  skip_exponent  ( end beg')
   -  -1 =          \ if it was properly formed, beg' = end + 1
;

: $ends   ( beg len -- end beg)      \ convert $info to pointers
   OVER  +  1-  SWAP  ;

: >FLOAT  ( c-addr u -- flag) ( f: -- r | <nothing>)
   $ends     ( end beg)
   2DUP  fp#?  IF
      get_sign  ( sgnflag end beg')
      mantissa  ( sgnflag power end beg'') ( f: mantissa)
      exponent  ( sgnflag power')
      10 S>F    ( sgnflag power')          ( f: mantissa 10)
      f^n       ( sgnflag)                 ( f: mantissa 10^power')
      F*        ( sgnflag)                 ( f: |r|)
      IF  FNEGATE  THEN                    ( f: r)
      TRUE                                 \ successful conversion!
   ELSE  2DROP  FALSE                      \ not properly formed
   THEN
;
```

The example subroutine—nearly three pages long and viciously unclear—is the programming equivalent of a run-on (and on and on and on and…) sentence. Just as we eschew run-on sentences in the name of clear prose, we must do something about prolix subroutines. The proposed alternative breaks the main subroutine down into manageable pieces, defines them as separate subroutines, then recombines them. Note that the main subroutine, `>FLOAT`, exhibits exactly *two* control structures, and they are *not* nested. Hence there is no convoluted logic and no problem with debugging. Manifestly the subroutines `mantissa` and `exponent` will contain control structures—probably `BEGIN`… `WHILE`… `REPEAT` indefinite loops—but they need be neither complex nor nested. And these complexities are appropriately hidden from the main subroutine, that only cares whether its components do their jobs properly.

Because Forth programs directly access the cpu stack, they avoid the overhead from building up and tearing down stack frames incurred in languages like C, Fortran, Pascal and Basic. Thus fine-grained decomposition into short subroutines degrades performance much less in Forth than in other languages. C programmers concerned with performance therefore tend to write run-on subroutines. Experienced Forth programmers tend to write short, simple subroutines. My guess is that the example program was written by a C programmer, mentally translating directly to Forth without rethinking his coding style.

To conclude, there is nothing inherently unclear about any of the languages in current use despite the existence of reams and blivets[10] of unreadable code written in them. Well-commented, transpar-

ent code is an achievable goal: the key is to worry about performance *last* and write the clearest program one can.

3. Testability and safety

A digital computer is a form of deterministic finite state machine. When we add software, the number of effective states grows so rapidly with program size that it can be very difficult to say definitively, for a moderately complex program, that all possible responses to all possible inputs are known[11]. Thus the issues of testability and safety are closely linked.

I cannot emphasize too strongly that in a world where more and more routine tasks are delegated to computers, it is vital to be sure that both the systems and their programs are as safe as possible. The term *safe* is fairly elastic: it can mean preventing incorrect, unexpected, damaging or life-threatening outcomes; as well as protecting networked systems from access or attack by unauthorized persons. Although the latter has become a major issue in an era of computer viruses and misuse of private data, we shall have little to say about it. Instead we concentrate on how to write correct programs that either function as expected or fail gracefully.

An incorrect program can produce effects ranging from loss of data (tolerable if it is merely tomorrow's homework assignment, serious if it is your life savings); loss of an expensive piece of machinery (as in the recent Mars Climate Orbiter fiasco[12]); or even loss of life. Although disasters are more likely from wrong programs on computers used in control applications (vehicles, nuclear reactors, radiation therapy), even programs for data acquisition and analysis can cause casualties in the right circumstances. A case cited by Neumann[13] describes how software that collected wind-tunnel data, software that theoretically analyzed the aerodynamics, and software that analyzed low-speed flight tests on a new type of aircraft, all agreed that tail vibration would not be excessive. On the first test flight the tail broke off, crashing the plane and killing the crew. The probability that all three programs would contain independent errors that produced incorrect, but agreeing, results was small, but certainly not zero—it happened!

Since it is theoretically impossible to determine with certainty (within the lifespan of the programmer!) the behavior of complex of software running on a given computer, the best we can do is first,

---

10. Look it up! (Try the *OED* or *Dictionary of American Slang*.)
11. This is an example of the "halting problem" for a given program. See, *e.g.*, R.P. Feynman, *Feynman Lectures on Computation*, ed. by A.J.G. Hey and R.W. Allen (Addison-Wesley Publishing Co., Inc., Reading, MA, 1996), p. 80ff.
12. On 23 September 1999, a space probe intended to orbit about Mars crash landed because the attitude thrusters were calibrated in pounds but the NASA team assumed the calibration was in Newtons.
13. P.G. Neumann, *Computer-Related Risks* (Addison-Wesley Publishing Co., Inc., Reading, MA, 1995), p. 220ff.

estimate failure probabilities; and second, try to provide graceful ("failsafe") ways for a program to fail.

The easiest way to make software reliable is to keep it simple. This does not mean it is impossible to create a complex program—such as one that operates a multipurpose robot or flies an airplane—but rather that it is vital to break such programs up into small, manageable chunks that can be tested in isolation. A small subroutine can be understood completely, and its responses to all possible inputs established through systematic testing. Although when such subroutines are combined into larger programs their reliability becomes less certain, nevertheless this is our best hope for creating reliable programs.

The most common sources of program failure are address errors and logic errors. An address error—caused, for example, by attempting to access an array element with the index out of range—can lead to writing data into memory where program code is expected. When the computer attempts to interpret the data as instructions, there is no telling what will happen next. Some programming languages build in error checking to prevent compiling programs with addressing errors, with a notable lack of success. The notorious "General Protection Fault", too often encountered by users of Microsoft Windows®-based programs, results from addressing errors. Because of the difficulty of preventing addressing errors in software, Intel and other manufacturers have built into their cpu chips a "protected mode" of operation that allocates memory segments with different access privileges for program and data. When a program attempts to store data in memory with the wrong privilege level, the program fails, generally not gracefully. Usually the only remedy for a program that has caused a General Protection Fault is to kill it, losing any data that was being worked on, possibly damaging key files, and so on.

Logic errors generally result from excessively convoluted control flow or decisions nested excessively deeply. Too many programmers implement decisions *via* complex binary logic trees when some other form of decision structure—such as a jump table or finite state machine—would be more appropriate. Deeply nested binary trees have the unfortunate characteristic that, unless the programmer is very careful (and most are not!) the conditions that lead to a particular outcome are not mutually exclusive. In that case the logic becomes impenetrable. Sometimes such practices create branches that *no* condition can reach. Such branches are called "dead code": programs containing them are not exactly the acme of computer science.

The reliability of software is best established by testing. Systems and languages that make it possible to run more tests per unit time, or that facilitate incremental testing, should be preferred to systems that hinder testing or that permit only large chunks of code to be tested at a time.

Finding the bugs in a large complex program is obviously harder than finding the bugs in small pieces of code. For this reason I favor interactive languages like Forth, Lisp or Basic over noninteractive ones like C, Fortran or Pascal, that require a subroutine to be compiled with an exercise program in order to test it. With interactivity the feedback is much more immediate and the results of changes easier to analyze.

There are several key aspects of software testing. The most basic has to do with having an adequate supply of test cases with known results. The more such cases one can run, the more one can rely on the program for the cases one actually wants to compute, that have not been done before.

I have already emphasized the discipline of coding in small manageable units. Let me expand on this. If a given piece of code has one entry point and one exit point—that is, if the code is *structured*[14]—the flow of control is greatly simplified. The analysis of possible outcomes for given inputs is thereby simplified also. Some languages, notably Fortran and assembly language, permit multiple entry points to a subroutine. The reason for this is that overlapping the code can be a great space saver, something one may have to consider when memory is tight. Such an overlapped Fortan subroutine might look like this:

```
SUBROUTINE FOO(A,B,C)
   do some stuff involving C (and possibly A and B)
ENTRY: FOO1(A,B)
   do some other stuff involving B (and possibly A)
ENTRY: FOO2(A)
   do stuff involving A only
RETURN
END
```

If the calling program said `CALL FOO(A,B,C)` the program would execute everything (that is, the `ENTRY:` statements would be ignored), whereas if one said `CALL FOO1(A,B)` the program would be entered at the point `ENTRY: FOO1(A,B)` and continue from there. Multiple `RETURN` statements could also be interlarded in the code, giving rise to a multiplicity of entries and exits. (Lest the reader believe one cannot simulate multiple entry/multiple exit code in Basic or C, Basic's assigned `GOTO` or C's `SWITCH` statements offer unlimited scope for folly.)

Multiple-entry, multiple-exit code must be avoided at all costs. It is nearly impossible to test adequately and can lead to exceptionally elusive bugs. But is this asking the impossible, *i.e.* can we always avoid unstructured code? In my opinion every computational problem can be solved with a structured program. Structured code may require more memory, but it can be understood and debugged.

Certain disciplines have been invented to systematize the debugging and testing process. The simplest is *version control*. Here we assign to a given version of a program a number such as 0.34. We keep a copy of this program in a sacrosanct archive—preferably on a removable medium like a tape or diskette. The working copy resides—and gets modified—somewhere else. The program's assigned number has the following significance: the zero (0) means it is a test version, not to be released or published. The three (3) following the dot (.) means it is our third major attempt. We do not change the version number unless the current version is so buggy, or performs so poorly, that a major rewrite

14. "Structured" programming is a notion introduced by N. Wirth, the inventor of Pascal.

is indicated. The rewrite may entail changing algorithms or data structures, altering program flow, or some other change so momentous it severs the connection with earlier major versions of the program.

The four (4) following the 3 indicates that minor bugs have been found and corrected four times. We increment this number each time we discover, repair, and retest for a bug.

A second discipline that has proven useful is the assertion[15], already alluded to above. Assertions are statements about a program or subroutine. For example, suppose we have written a function BSEARCH that locates items in an ordered list by the binary search algorithm[16]. Some languages allow a program designed to exercise and test BSEARCH to contain statements like

```
ASSERT( list( BSEARCH("item", list) ) = "item" )
ASSERT( list( BSEARCH("item", list) - 1) = "" )    ' returns the empty string
ASSERT( list( BSEARCH("non-item", list) ) = "" )   ' returns the empty string
```

which determine whether the routine will find an item contained in the list, that it is the first occurrence of that item, and that if it looks for something not in the list it will indicate failure in an appropriate manner.

Forth, C and C++ permit the definition of macros[17] that can perform this sort of assertion. For example in C,

```
#define ASSERT(e)  if (!(e))  error("assertion failed")
   // e stands for "any expression"
```

In Win32Forth assertions are defined as

```
CR .( Loading Assert Wordset...)

0 value assert?

: assert(       ( -<words>- )
                assert? 0=
                IF      POSTPONE (
                THEN    ; IMMEDIATE
```

---

15. J. Bentley, *ibid.*
16. We suppose BSEARCH returns the index in the list, of the first occurence of the desired item.
17. A macro ("macro-operation") is an instruction to compile a certain set of operations under a common name. It is something like a subroutine, except that instead of the program branching to, and returning from, a separate piece of code, the macro is replaced by its corresponding instructions in-line in the main program.

```
: ?assert       ( flag nfa -- )
                SWAP
                IF      DROP
                ELSE    CR ." Assertion failed in " NFA-COUNT TYPE
                        CR ." Enter to continue, ESC to abort"
                        KEY 0x1B  =  IF  ABORT  THEN   CR
                THEN    ;

: )             ( -- )
                ?COMP
                LAST @ POSTPONE LITERAL
                POSTPONE ?assert ; IMMEDIATE
```

An example of usage is

```
TRUE TO assert?

: atest         ( -- )
                10 0
                DO      i .
                        assert( i 5 < )
                LOOP    ;

atest 0 1 2 3 4 5
Assertion failed in ATEST
Enter to continue, ESC to abort
  ok
```

Often one does not need to go as far as actually defining and inserting such macros. Comments describing what the state of the program should be at given points may well be enough, especially if the program can be run in a single step mode *via* a debugger[18].

It is useful to specify "loop invariants" for looping constructs. That is, a loop is just a sequence of actions that gets repeated either a definite number of times or until some condition is satisfied. The code within the loop must find the same number and kinds of inputs, and produce the same number and kinds outputs, every time the loop is executed. The list of inputs is a loop invariant, as is the list of outputs. Another sort of loop invariant can be an assertion about inequality. That is, the current value of some variable must—assuming the program is correct—always remain bounded above and below by limits `U` and `L`. If so, state it as a comment or an assertion.

18. Debuggers are programs that—depending on their degree of elaboration—run your program as a subroutine, executing one line at a time (or executing up to some *breakpoint*) then pausing to let the programmer inspect the current values of variables and so on. As a Forth programmer I rarely use a debugger, but they are necessities of life for testing Fortran, C, Basic, and especially assembly language programs.

4. Maintainability

Programs sometimes have surprisingly long lifetimes. Morsels of code we write to solve one problem often end up in libraries—often without much editing for readability or code quality[19]. Scientific programmers encounter the problem of program maintainance in two contexts:

- they find they need to re-use (with modifications) code they themselves wrote six months earlier, that was to have been a one-off; and find they can neither understand what it does nor how it does it;
- they want some code from a library to dosome particular task, and find that it needs some minor tweaking, which cannot be done without understanding thoroughly the algorithm and code.

Even if it is your own code, when you look at it for the first time in six months or a year it will seem foreign. Hence in either instance one must try to understand a program written by a stranger, probably one with a decidedly odd and uncouth programming style, and always with completely inadequate documentation.

Adhering religiously to three rules will make programs maintainable:

- Avoid tricks and “hacks”.
- Hide data.
- Design for change.

Hacks

What do we mean by “avoid tricks”? We previously saw a subroutine that used the locution

```
$F0  AND  2/  DUP  2/ 2/  +
```

for getting the tens digit of a 2-digit packed BCD integer, and converting it to binary—in lieu of the equivalent

```
4 RSHIFT   10 *
```

which would have been much clearer.

19. Commercial and non-commercial purveyors of code libraries deal with safety and testing issues with *disclaimers*: notices in legalese stating bluntly that the user is on his own, *caveat emptor*. I have yet to see the disclaimer that warrants that the program will do **anything** useful, much less that it will refrain from doing something harmful. The disclaimer in *Numerical Recipes* reads “We make no warranties, express or implied, that the programs contained in this volume are free of error, or are consistent with any particular standard of merchantability, or that they will meet your requirements for any particular application. They should not be relied on for solving a problem whose incorrect solution could result in injury to a person or loss of property. If you do use the programs in such a manner, it is at your own risk. The authors and publisher disclaim all liability for direct or consequential damages resulting from your use of the programs.”

Here is another simple and (I blush to say it) cute trick I use frequently in my own code. But when I publish code I am careful to extirpate the trick. I will first show the trick, then explain why the best advice is "don't".

Modern (ANS-compliant) Forth systems represent the `TRUE` flag as an integer register with all its bits set to 1, and the `FALSE` flag as all bits set to 0. If the cpu uses 2's-complement arithmetic[20] `TRUE` has the same representation as the integer –1 and `FALSE` is the same as the integer 0. Suppose we want to skip a certain character in a string (for example, my FORmula TRANslator that embeds Fortran expressions in Forth subroutines skips a leading "–" sign). This can be accomplished by advancing a pointer one unit if it currently points to the character we want to skip, and by not advancing the pointer otherwise. The usual solution for such an action would use an `IF` statement, as in

```
: skip    ( ptr char -- ptr+1 | ptr)
   OVER  C@     \ duplicate pointer and get character
   =            \ is it equal to char?
   IF  1+       \ if so, add 1 to ptr
   THEN
;
```

But the special nature of the `TRUE` flag (on a 2's complement machine) allows us to *compute* rather than decide the action:

```
: skip    ( ptr char -- ptr+1 | ptr)
   OVER  C@     \ duplicate pointer and get character
   =            ( -- ptr -1|0)
   -            \ ptr = ptr - flag
;
```

However, on a 1's-complement cpu (they are no longer plentiful, but they do exist) setting all bits to 1 yields the largest integer possible. It would not be a good idea to subtract *that* from the pointer. In other words, my `skip` subroutine is not fully portable. As the ANS94 Forth Standard requires us to say, it creates an "environmental dependence" (because it assumes 2's-complement arithmetic). This can be fixed (if we want to do it) by rewriting it as

```
: skip    ( ptr char -- ptr+1 | ptr)
   OVER  C@     \ duplicate pointer and get character
   =            ( -- ptr flag)
   1 AND        ( -- ptr 1|0)    \ bitwise AND
   +            \ ptr = ptr + (1 & flag)
;
```

While correct, portable and still decision-avoiding, the locution

```
OVER C@  =  1 AND  +
```

20. See Chapter 1. Oddly, despite its simplicity and utility, 2's-complement arithmetic is not universal. Some processors use other methods such as 1's compement or signed magnitude arithmetic.

has lost its magic. Better to avoid it altogether since the version using `IF` is self-explanatory.

Here is another example, from Fortran this time. Before Fortran permitted dynamical array allocation (that is, passing an array's address to a subroutine rather than all the array elements) some programmers had discovered the following trick:

```
   PROGRAM MAIN
   DIMENSION A(100,100)
      ……
   CALL HILBERT(A(1,1), 20)
      ……
   END

   SUBROUTINE HILBERT(A,N)
   DIMENSION A(1)
   DO 20 I=1,N
      DO 10 J=1,I
         A(I,J) = 1./(FLOAT(I+J)
         IF (J.LT.I) A(J,I) = A(I,J)
10    CONTINUE
20 CONTINUE
   END
```

This trick depended on knowing a number of details about the compiler—in particular, that matrices are stored column-wise in contiguous memory cells, and that the element `A(1,1)` actually represents the base address of the array. It also depended on knowing that the (variable) arguments in a subroutine's parameter list were the addresses of these items in the calling program. Since some Fortran compilers routinely passed values rather than addresses, the trick was guaranteed to fail when compiled with the latter. Finally, it assumed the compiler would not notice that the dimensionality of `A(1)` (in the `DIMENSION` specification) was not the same as that of `A(I,J)` used elsewhere. If the compiler checked for consistency, the trick would be worthless.

Here is a third kind of trick to avoid. Unlike the others it is completely portable. What is wrong with it is that it is so subtle (albeit fast) that some analysis is required to see why it works. Worse, it employs "magic" numbers whose significance to the algorithm is by no means clear[21].

```
\ count the number of non-zero bits in an unsigned 32-bit integer
OCTAL     \ interpret integers as base-8 until further notice
: ones ( uint32 - numbits)
    DUP 1 RSHIFT     33333333333 AND
    DUP 1 RSHIFT     33333333333 AND  + -
    DUP 3 RSHIFT  +  30707070707 AND  77 MOD
;
DECIMAL   \ restore base-ten arithmetic
```

21. I learned this trick several years ago from Michael Pruemm. (In fact it is a very good trick, but it does not lead to maintainable code.) The ANS Forth subroutine `RSHIFT` is equivalent to the C function `>>`.

A more maintainable way to do it is the bit table method advocated by Bentley[22]: we create a table recording how many bits are in each of the integers in the range—say—[0,15] (that is, a table of bits per nybble). Then we successively get the least significant nybble, retrieve its bit count, add it to the net count, shift the integer rightward 4 bits, and repeat (8 times in all). The code for this is

```
: under+    ( a b c -- a+c b)  ROT  +  SWAP  ;
: bits                                     ( n -- count )
    0 SWAP
    BEGIN  ?DUP WHILE
        DUP   1-  AND   1 under+
    REPEAT
;

CREATE  BITS/NYBBLE    \ make a 16-element table
     0 bits  C,   1 bits  C,   2 bits  C,   3 bits  C,
     4 bits  C,   5 bits  C,   6 bits  C,   7 bits  C,
     8 bits  C,   9 bits  C,  10 bits  C,  11 bits  C,
    12 bits  C,  13 bits  C,  14 bits  C,  15 bits  C,

: ones  ( uint32 -- numbits)
    0 swap              ( numbits=0 uint32)
    7 0 DO              ( repeat 7 times)
        DUP
        [ BINARY 1111 DECIMAL ] LITERAL  AND    \ get rightmost nybble
        CHARS                                   \ convert to offset
        bits/nybble +  C@                       \ retrieve table elt
        under+                                  \ add to numbits
        4 RSHIFT                                \ shift 4 bits to right
    LOOP
    CHARS  bits/nybble +  C@  +                 ( numbits)
;
```

Although considerably lengthier than the tricky algorithm, this code is much clearer. To emphasize the point that we are getting the last 4 bits each time, the AND operation is exhibited with the binary representation 1111 of (decimal) 15 or (hex) FF. (In C or assembler we would simply say `1111B`.) If we are willing to waste some more space we can use 256 bytes instead of 16, to hold a table of bit-counts for the first 255 integers. Then we would AND with 255 (`11111111B`) and right-shift 8 places, repeating 4 times. That is, we can trade space for speed.

The moral of these examples is that most programming languages permit the use of clever tricks, but it is wiser to avoid them if maintainability is our goal.

Data hiding

We turn now to "data hiding" or "information hiding" as a principle of program design. At its most elementary level, if a subroutine needs a "magic" number (such as octal 33333333333 or 30707070707) the number should be built into that subroutine as a literal. It never needs to appear

22. *ibid*, p. 83.

or be referenced elsewhere in the program, so it need not reside in common storage. Thus it is less easily corrupted by another subroutine—it is hidden.

Hiding information can also be part of designing for change. For example, if the state-of-the-art display has 640 by 480 pixels, rather than sprinkle literal 640's and 480's around randomly in your graphics programs like raisins in raisin bread, it is better to define constants

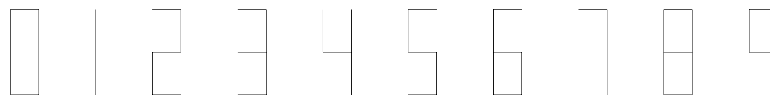
```
640 CONSTANT screen_width
480 CONSTANT screen_height
```

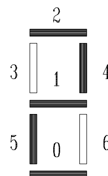
(or the equivalent in your language of choice) at *one* place in the program. Then if the new top of the line displays have 1280 by 960 pixels (for example), only one change is needed. Better still, relegate all such data (on the display's salient characteristics) to a separate file to be read by the subroutine that initializes the display parameters for the current hardware.

It would be equally unwise to rely on a particular coding convention for—say—alphanumeric data. Today the common usage in Western countries is extended ASCII[23]; however, at one time EBCDIC[24] was a rival standard, and Unicode (using 2^{16}=65536 rather than 2^{8}=256 character codes) may well become the standard of the 21st Century.

In other words, within your program numerical constants should be referenced by name, not by literal value. To do otherwise is as bad as writing an organizational instruction manual that refers to Jennifer, the person in charge of the Help Desk—the instant she leaves the company, the manual becomes obsolete. Obviously one should refer to the "Help Desk Manager"—then a change of person or sex will require no updates.

Here is another example where information hiding both increases the readability of a program and facilitates change: suppose you wanted to operate a 7-segment display device like the one below. By turning on various segments one can display the digits 0–9 as shown. For example, with segments 0,5,1,4 and 2 turned on the device displays the digit 2. One could obviously program this with an array of 8-bit integers:





23. American Standard Code for Information Interchange
24. Extended Binary Coded Decimal Interchange Code

```
DIM a(10)
a(0) = 01111101B
a(1) = 01010000B
    …
a(9) = 01041110B
```

However this requires the programmer to translate in his head the conversion of which segments are turned on to a binary representation of an integer.

It would be much kinder, easier to maintain, and more change-friendly to say (I'll do it in Forth, but it will be similar in any language)

```
\ A simple program to drive a segmented display
   7 CONSTANT  #segs                         \ number of segments

   : install,   ( n0 n1 n2 n3 n4 n5 n6 --)    \ the digits can be in any order
      0                                      \ initialize the array element
      #segs 0 DO    SWAP  DUP                (  n's 0 n)
                    0 #segs 1+ WITHIN   NOT   \ check if digit seg_# in range
                    ABORT" a segment number is out of range"
                    DUP  #segs <              \ is it a segment # ?
                    IF  1 SWAP  LSHIFT   THEN  \ move the bit into position
                    +                         \ add the bit to the result
      LOOP
      ,                                      \ store result in next cell
   ;

   CREATE  display              \ the integers are the segments turned on
                                \   -- 7 is a "don't care" digit
      7 0 5 6 3 4 2 install,     \ 0
      7 7 7 7 7 4 6 install,     \ 1
             …
      7 7 1 3 4 2 6 install,     \ 9

   : >display   ( n --)   CELLS  display +  @    >device  ;
```

Let us note what this code accomplishes:

- The only number in the program is the number of segments in the display, defined as a `CONSTANT` so if it is changed, the change propagates through the program with no further effort. This minimizes the chance for error if the code must be moved to another machine, modified to run a different segmented display (for example with more segments), or otherwise changed over time.

- The information about the device itself is contained in the table showing which segments get turned on for a given number. If a different device is to be used, its table will have to be constructed by hand. To make it easier for the code maintainer, the integers do not have to be entered in any particular order.

- To make the table uniform and easy to read, we enter a "dont care" number as a placeholder; the program ignores this number when encountered. Since the actual segment numbers range from `0` to `#segs - 1`, we can use `#segs` itself for the "don't care" entries.

- The virtue of passing a fixed number of arguments (including "don't care" 's) for the subroutine `install,` —that constructs each entry in the array `display`—is that we can use a definite

loop. Routines that must cope with varying numbers of arguments are difficult to write and debug (it is easier in Lisp, a language designed to cope with such constructs as variable-length argument lists). Opt for simplicity whenever possible (and mostly it *is* possible).

Some noteworthy additional points: some manuals of programming style advocate putting all numerical constant definitions in a single section of the program. This is not a bad idea for languages that are batch-compiled (C, Fortran) since then the programmer knows where to look for them. However, in a long program a given constant can be hard to find since the initialization section will also be long. If the language permits programming in modules, the repetition of names (that is, the routine `zorch` in Module `A` is distinct from `zorch` defined in Module `F`) renders this sequestration of initializing code into one section literally impossible. My preference, therefore, is that "local" constants—that is, constants particular to a module of code—be located physically with the module.

What about error handling? Some style manuals advocate putting all the error handling in one place rather than sprinkling it through the code. Again, this fights modularization and is just hard to do in any long program. I prefer to keep error handling within a module, or if it is simple, even within a subroutine (as in `install`, defined above). Here I am less doctrinaire since there will be cases—for example when a program has to be especially fault-tolerant so a machine can keep running—where the error handling has to be more sophisticated than simply causing an `ABORT` during compilation (as above).

Finis