A Description of One Programmer's Programming Style *Revisited*

Adam N. Rosenberg

1990 August 1
2001 October 1

ABSTRACT

We present the outlook of a programmer at AT&T Bell Labs who has written much code during his eight years *there and thirteen years in other places.* This document describes the author's own programming style and what he considers important in generating reliable, readable, and maintainable code.

Since this document is the opinions and prejudices of an individual, it is written in the first person in a conversational tone, and with subjects covered in no particular order. It is intended to be a repository of good questions rather than a source of answers.

The author feels that many programmers suffer from gross inattention to detail in writing code. He suggests that clarity and consistency will be rewarded sooner than most people think.

A veteran of many languages and operating systems, the author today finds himself using the MS-DOS and UNIX® operating systems and the FORTRAN and C languages. The examples here are all either FORTRAN or C.

A decade later the author feels that programming productivity in our "post modern" world has decreased *sharply from 1990 to 2000. Many of the reasons for this decline were discussed in the original version of this paper. Our pleasure in prescient prognostication is mitigated by frustration with the "dumbing down" of computing in general. Based on this unhappy downturn of programming education and style, we have added materal (in italics) emphasizing areas of recent concern. The original text and examples (still in regular type) have been left virtually intact.*

WARNING - THIS DOCUMENT CONTAINS PERSONAL OPINIONS!
Do not be alarmed if you disagree with something in it.

Contents

WARNING - THIS DOCUMENT CONTAINS PERSONAL OPINIONS!
Do not be alarmed if you disagree with something in it.

WARNING - THIS DOCUMENT CONTAINS PERSONAL OPINIONS!
Do not be alarmed if you disagree with something in it.

A Description of One Programmer's Programming Style *Revisited*

Adam N. Rosenberg

1990 August 1
2001 October 1

1 INTRODUCTION

After spending a few years writing a lot of computer code, I have seen software migrate from a passive role serving active users to an increasingly active role serving more passive users. Further, software being written is woefully lagging behind the needs of its users in function and reliability. More and more I hear that some program just doesn't solve the problem it was written to solve.

Some of this is a matter of expectations. If a program works halfway well, then we can count on somebody trying to use it where it was never intended. When the program performs successfully, then we hail it as a triumph of modern software. And when it fails, then we berate the user for not understanding the product.

But much of today's failure in software has nothing to do with user expectations. We have zillions of lines of code that does nothing well. Maybe it's a spreadsheet, maybe it's a financial planning system, and maybe it's a character-based video test system, but you are never going to tell by running it! For every well written, superbly documented, and rock-solid reliable example like LOTUS-123, there are more bad examples than any of us would like to admit.

For those unlucky software developers who just have no knack for writing code, there is nothing I can do. I know otherwise clever people who can't seem to get the program written and debugged to solve a standard quadratic equation using the standard quadratic formula.

WARNING - THIS DOCUMENT CONTAINS PERSONAL OPINIONS!
Do not be alarmed if you disagree with something in it.

```
      program quad
c
      write (*,*) 'Hello, I am a computer.'
      write (*,*) 'I solve quadratic equations.'
      write (*,*) 'Give me A, B, and C --> '
      read  (*,*)  a, b, c
c
      if (a.eq.0.0) then
        write (*,*) 'Your value of A is zero.'
        write (*,*) 'Your equation is not quadratic.'
        stop
      end if
c
      disc = (b*b)-(4.0*a*c)
      if (disc.gt.0.0) then
        root1 = (-b-sqrt (disc))/(2.0*a)
        root2 = (-b+sqrt (disc))/(2.0*a)
        write (*,10) root1, root2
10      format ('Roots are ',f15.4,' and ',f15.4,'.')
      else if (disc.eq.0.0) then
        root = -b/(2.0*a)
        write (*,20) root
20      format ('Double root is ',f15.4,'.')
      else
        real  = -b/(2.0*a)
        aimag = sqrt (-disc)/(2.0*a)
        write (*,30) real, aimag
30      format ('Roots are ',f15.4,' +/- ',f15.4,' I.')
      end if
c
      stop
      end
```

Let me suggest that programmers should write code with the expectation that it really has to work even when the programmer isn't around to explain it or fix it. Let me add the expectation that this code should be useful years down the road, not only as an executable file but as the basis for an improved version of the product. And let me further add the expectation that the same programmer who is writing the code should expect to be the programmer who has to add these enhancements in a timely way.

Old style hackers who wrote software for the original NASA during the Mercury, Gemini, and Apollo programs wrote code this way. The men and women who designed the first electronic switching systems wrote code this way. The programmers who wrote the operating systems and compilers that once set the standards for an industry wrote code this way. And you can write code this way as well.

WARNING - THIS DOCUMENT CONTAINS PERSONAL OPINIONS!
Do not be alarmed if you disagree with something in it.

As a veteran of a great deal of code doing a great many different things, I have learned a few lessons. Several have written books telling you how they think you should write your code; I am not doing that here. Instead, I am telling you how I write my code and what kinds of issues I have had to think about along the way. If you deal with all the issues I raise, whether or not you deal with them my way, then I expect your code to meet the old style expectations I described above.

Several recent social changes have generated a sense of urgency. The pressure for conformity has increased enormously, young people are more willing to believe authority, and accountability to success has decreased. The result of these is that a curriculum without demonstrated success is becoming the unbreakable mold of a generation of programmers intolerant of alternative approaches to programming problems. That I do not subscribe to these brave new ideas makes me more concerned about the new level of intolerance. That these new ideas accompany a dramatic decline (page 75) makes me skeptical of them and increases my resolve to do the programming jobs that need to be done in a professional and responsible manner. I am increasingly put on the defensive for my suggestion that there is another way to write programs than these new ideas which are becoming universal and which have failed dramatically.

2 STANDARDS

I hate porting code from one environment to another. There are few insurance policies in this arena, but one way I can almost always blame somebody else for my misfortune, is to stick to the standard. There are lovely enhancements provided by thoughtful folks at IBM, DEC, Microsoft, WATCOM, and AT&T. I use few of these and fully expect to be bitten hard someday for those I use.

I don't mix languages in a single application except at gunpoint. The code is almost never portable to anything else and the resulting errors are almost always hard to debug. In general, I use FORTRAN for most things but if I need recursion, structures, bit-twiddling, or memory management, then I use C. (Even my C code, however, tends to look like FORTRAN.)

In general, I put parenthesis in to denote order of operations when left to right would get the wrong order. For readability, I'll write

```
    h = b*10+c
```
without parentheses but
```
    h = (a*100)+(b*10)+c
```
with parentheses.

2.1 FORTRAN

With two flagrant exceptions, I use the ANSI X3.9 - 1978 FORTRAN Standard. It allows for assignment, comparison, and concatenation of character strings and substrings. It allows formatted input from strings and formatted output into strings. It has the IF-THEN-ELSE construct. And it has the PARAMETER statement.

The first feature of every compiler I use is that a tab is the equivalent of indenting past card column six. I know hard-core FORTRAN programmers who tap the space bar six times for each line, but every executable line in my code starts with a tab or has a tab after the numeric statement label. Continuations are spaced out to column six and numbered 1, 2, 3, ...

The second feature of every compiler I use is the INCLUDE statement.

```
INCLUDE 'PROGRAM.HDR'
```

tells the FORTRAN compiler to open PROGRAM.HDR during compilation and compile the contents of PROGRAM.HDR where the INCLUDE statement is. INCLUDE statements can be nested so that one included header file can include another. I often have a PROGSIZE.HDR file included in my PROGRAM.HDR file that has all the parameters for my array dimensions.

There are certain legal FORTRAN statements I avoid. I never use arithmetic IF statements

```
IF (A-B) 20, 30, 40
```

or computed GOTO statements

```
GOTO (110, 120, 130, 140, 150) N
```

in my code. Instead of using LOGICAL variables I usually use INTEGER variables set to `ITRUE` (one) or `IFALSE` (zero).

There are several neat enhancements I never use because some compilers don't support them and sooner or later I will find myself running on one of them. Reading and writing unformatted data from and to character strings are not part of standard FORTRAN. Nor is the DO WHILE statement, the DO UNTIL statement, or the END DO statement part of standard FORTRAN. Some compilers support the RECURSIVE keyword, some make all subroutines allow recursion, and some do terrible things when a subroutine calls itself. The same story applies to structured variables (with the dot in the variable name). If I must have recursion or structures, then I use C.

2.2 The C Programming Language

There is a new ANSI C standard which is, of course, not compatible with the standard C everybody uses. *(One* big *improvement in the last ten years is that ANSI C has become nearly universal in use. I have more to say about that on page 6.)* I happen to agree with the ANSI folks that some of the original decisions made in the C language were awful enough that they should be changed and it is worth the trouble to program around the differences.

I insist on prototyping subroutines and functions. If a function F returns a FLOAT value from two INTs and a CHAR pointer, then put

```
float f (int, int, char *);
```

at the top of the source file or in some global header file if there are multiple source files.

I suppose I will break down and use ANSI style declarations

```
float f (int istart, int istop, char *pstring);
```

when I stop using the MICROSOFT 4.00 C compiler (on my DOS personal computer) which does not support them.

I strongly avoid structures since I find code hard to read with them and I tend to prefer IF-ELSE constructs to case statements. Let me offer a word of warning to the neophyte C programmer who shares my affinity for IF-ELSE constructs. Like any FORTRAN hacker, I indent my IF-ELSE constructs the way I think of them.

```
if      (i == 1)
  do_first_case ();
else if (i == 2)
  do_second_case ();
else if (i == 3)
  do_third_case ();
else if (i == 4)
  do_fourth_case ();
else
  do_other_cases ();
```

But the C language has no ELSE IF statement like FORTRAN. Rather the compiler is reading your code this way

```
if (i == 1)
  do_first_case ();
else
  if (i == 2)
    do_second_case ();
  else
     if (i == 3)
       do_third_case ();
     else
       if (i == 4)
         do_fourth_case ();
       else
         do_other_cases ();
```

and will typically generate stack-overflow messages during compilation with a long enough list of else-if constructs.

The “official” standards for C allow for compiler directives to start with a pound sign (#) in column one followed by the directive or some white space and the directive. The compilers I use allow white space before the pound sign as well. So if my compiler directive is made clearer to the reader by indenting it, I do just that. (For the one time I used a compiler that required the pound sign to be in column one, I wrote a script for my editor that moved the pound sign to column one and put the directive in the indented space.)

I have two standard mistakes that consume much of my C debugging time. First is forgetting an ampersand in some kind of SCANF call. SCANF, FSCANF, and SSCANF require all their arguments to be pointers. The second mistake is putting one equal sign (=) inside an if statement instead of two (==). Because the second case is so troublesome, I avoid any construct with a single equal sign in the decision part of an IF, WHILE, or FOR statement.

There are many aspects of C that some find hard to read. I know people who would use abbreviations like these.

```
#define BEGIN   {
#define THEN    {
#define END     }
#define END_IF  }

#define EQ    ==
#define NE    !=
#define LT     <
#define LE    <=
#define GT     >
#define GE    >=

#define FOREVER    while (1)
#define ever       (;;)
#define DO(j,m,n)  for (j = m; j <= n; j++)

#define INCREMENT  ++
#define DECREMENT  --

#define CALL
```

While the initial impact of abbreviations like these is to make it easier to read a particular piece of code, the long term effect is to create a special version of the C programming language that only one person can use. Even if I learn to read somebody's special version of C, it will still take me forever to learn to write it. I feel it is best to use the language in its standard form, more or less, and stay in touch with other programmers in the C community.

In the last ten years, ANSI C has become universal and it is a major *improvement over earlier versions. (I don't know if SUN has an ANSI C compiler yet, but I believe the GCC compiler from the Free Software Foundation[1] is available for the SUN computers.) In addition to cleaning up function prototypes, the ANSI folks really cleaned up the library functions. Many compilers had little differences in what library functions would do or what they would return.*

My personal example of an ANSI C fix is FPRINTF, the file print function, which returns the number of characters printed. When the disk is full, FPRINTF returns EOF, but some pre-ANSI compilers returned zero instead. Testing for a full disk when writing *a file really should be standard and now it is.*

3 SPACING

There are many standards for spacing and often one piece of code seems to follow many standards. I emphatically emphasize the need to select just one and here is mine.

First of all, I don't put spaces in variable names. C doesn't allow it and FORTRAN code starts to look silly when `NOTJ` is written `NOT J`. (I have no objection to underscores in C variables, however.) Certain keywords really look better with spaces in FORTRAN such as ELSE IF and END IF. I put white space on both sides of any keyword and between any function name and parentheses. I also put white space around assignments and logical operators in C.

```
=   +=   -=   *=   /=   &=   |=   ^=   ==   >   >=   <   <=   !=   &&   ||
```

I do not put spaces around FORTRAN logical operators since the periods already create a visual distinction.

```
.EQ.   .GT.   .GE.   .LT.   .LE.   .NE.   .AND.   .OR.
```

I put spaces after commas except in DO loops. I put no spaces around operators like +, -, *, and /, I put no spaces around parentheses for subscripts, and I put no space between subscripts. I put no other spaces in arithmetic expressions. In C, I double space after the logical expression in if statements and between multiple statements on the same line. I also add extra space to line up parallel constructions.

```
dimension fxol(100), fyol(100)
  . . .
gx = fx+(cos (theta)*fxol(jl))+(sin (theta)*fyol(jl))
gy = fy-(sin (theta)*fxol(jl))+(cos (theta)*fyol(jl))
if      (gx.gt.0.and.gy.gt.0) then
  h = max (fx, fy)
else if (fx.gt.0.and.fy.gt.0) then
  h = max (gx, gy)
else
  h = 0.0
end if
```

```
float fxol[100], fyol[100];
  . . .
gx = fx+(cos (theta)*fxol[jl])+(sin (theta)*fyol[jl]);
gy = fy-(sin (theta)*fxol[jl])+(cos (theta)*fyol[jl]);
if      (gx > 0.0 && gy > 0.0)  h = max (fx, fy);
else if (fx > 0.0 && fy > 0.0)  h = max (gx, gy);
else                            h = 0.0;
```

Breaking lines is often ugly, but I allow only 72 characters on a line (with tab spacing set to eight). This way lines don't wrap around on almost any editor and when I print the file with line numbers, it still looks nice. I never break in the middle of a keyword or name and try to break at the highest level practical. The last examples might have looked like

```
                                        gx = fx+(cos (theta)*fxol[jl])
                                              +(sin (theta)*fyol[jl]);
                                        gy = fy-(sin (theta)*fxol[jl])
                                              +(cos (theta)*fyol[jl]);
```

if they had been indented too far to fit on one line.

4 NAMES

The better the names of things, the less work the comments have to do. Often programmers accused of too few comments are actually using enough comments but unclear names of things.

When programs evolve the old names for things can become silly. I once had two constructs denoted by letters of the alphabet

```
c                    F = flying time
c                    T = elapsed time
```

so when airplane fleet type came along, I used

```
c                    P = plane type (fleet)
```

which made some sense at the time. When the distinction between flying time (F) and elapsed time (T) disappeared, the letter F was freed up and it behooved me to rename all the variables using P for plane type instead to use F for fleet. (I even wrote a program to facilitate this.)

It is worth some pain to select names carefully to begin with and worth even more pain from time to time as a program evolves to keep the names up to date with the functions. Obviously, there is a point of diminishing returns and code will always be out of date, but the time spent updating code will usually pay for itself many times over.

4.1 File Names

Different hardware and software support different sorts of file names, but almost everybody supports a structure with a file name of up to eight characters, a period or dot (.), and a file extension of up to three characters. The characters can be letters, numbers, or hyphens. I avoid multiple periods, names of nine or more characters, and extensions of four or more characters. If MS-DOS will take the file name, then so will almost everybody else. If the operating system allows for case distinction in file names, then I use lower case only.

Let me implore those using databases of any form to make file names as informative and suggestive as possible. I want to know what a file is and when it was created. I tend to name directories or files with six digit dates of the form YYMMDD. This way the dated names come out in chronological order. My files usually have descriptive extensions.

If my Airline Schedule Planner has airport files, fleet files, leg files, and hookup files for 1990 July 19, then I might name them 900719.APT, 900719.FLT, 900719.LEG, and 900719.HUP. They could be tied together by a Schedule Planning header file, 900719.SP. This way, if I change only the leg file next week, I can create a new header file 900724.SP that points the program to 900719.APT, 900719.FLT, 900724.LEG, and 900719.HUP.

The urgency of keeping to the "eight-dot-three" file name has only increased in recent years. Microsoft Windows allows the longer names but the DOS tools that do useful things and that really work *use the short names and the schism between SOURCEFILES.DAT and SOURCE~1.DAT seems to attack often and to bite hard.*

One change I have made in recent times is to use a three digit code for the date described in Section 11.5 (page 66). A six or eight digit date field uses too many of the eleven characters available for a file name. Names like APT-07K.DAT, FLT-07K.DAT, and LEG-07Q.DAT tells me that the leg file is a week newer than the others. For file backups I use the three digit IDATE field as the extension.

4.2 Variable Names

FORTRAN allows up to six characters in a variable name with letters and numbers only where the first character is a letter. Nine out of ten compilers allows up to 30 characters with dollar signs and underscores, but I don't take advantage of it. I still use a compiler or two that doesn't allow longer names.

In C the original standard was eight characters only, but *everybody* allows up to 30, so I do use longer names. However, the only names that are longer than eight characters in my code are pseudo-logical names used for flow control.

```
#define FALSE 0
#define TRUE  1
void adjust (int, float *, float *, int *);
float fxol[100], fyol[100];
int nothing_was_changed, ichange;
int jl;    /* line */
int numl;  /* number of lines */
 . . .
nothing_was_changed = TRUE;
while (TRUE)
{
  for (jl = 1; jl <= numl; jl++)
  {
    adjust (jl, fxol, fyol, &ichange);
    if (ichange)  nothing_was_changed = FALSE;
  }
  if (nothing_was_changed)  break;
}
```

In FORTRAN, if a variable is not explicitly declared, then the first letter determines its type. Further, one can define the implicit type of variables as a function of the letter. The FORTRAN default is

```
implicit integer (i-n)
implicit real (a-h, o-z)
```

and I amend this default by putting in my code

```
implicit character*1 (q)
implicit character*4 (p)
implicit character*32 (o)
implicit character*256 (z)
```

to define string variables by length.

Unlike FORTRAN, C has no implicit typing. Many people seem to see this as a golden opportunity to obfuscate their code by having no consistent notation for INT variables, FLOAT variables, pointers, CHAR variables, *et cetera.* I still use first letter I through N for INTs and A through H and S through W for FLOATs. I tend to use P for pointers, Q for CHARs, X for FILE pointers, and Q, Y, and Z for string arrays.

The C programming language lets one declare all kinds of strange variables. This should not be construed as license to do so. With a few exceptions, I keep variables as characters, integers, floating point (usually double precision), and arrays of these three things. Anything else is usually more distraction than help, especially to the poor soul trying to figure out what the program does. And that poor soul may be me in six months.

Someday I'll have a program so complicated that I'll have more than 25 distinct data categories, *which I call "primitives,"* but for now I use the individual letters of the alphabet (except O) to denote them. For example, consider a program that works on two types of data, circles and lines. Circles have an (X,Y) location and a color and lines connect circles to each other but have colors of their own. So I might start by defining

```
C = circle
K = color
L = line
X = X-coordinate
Y = Y-coordinate
```

with the kute abbreviation of kolor used since I find having C stand for circle more important.

Indices for circles would have the names `JC`, `KC`, and `LC` while indices for lines would have the names `JL`, `KL`, and `LL`. I use FX or GX to refer to floating point X-coordinates and `JX` or `KX` to refer to integer X-coordinates. My convention is to use `MAXC` and `MAXL` for the array size parameters for circles and lines and to use `NUMC` and `NUML` for the number being used in the problem being solved.

Now we define array names where the first letter is N for integer and F for floating point. The letter O stands for "of" so the name `FXOC` means "floating point X-coordinate of circle." The statements

```
      parameter (MAXC = 1000)
      parameter (MAXL = 1000)
c
      common fxoc(MAXC), fyoc(MAXC), nkoc(MAXC)
      common numc
c
c                fxoc(jc) = X-coordinate of circle JC
c                fyoc(jc) = Y-coordinate of circle JC
c                nkoc(jc) = color of circle JC
c                numc     = number of circles
c
      common ncol(MAXL), nccol(MAXL), nkol(MAXL)
      common numl
c
c                ncol(jl)  = first circle of line JL
c                nccol(jl) = second circle of line JL
c                nkol(jl)  = color of line JL
c                numl      = number of lines
```

declare a set of arrays in FORTRAN and

```
#define MAXC 1000
#define MAXL 1000

float fxoc[MAXC+1];  /* X-coordinate of circle */
float fyoc[MAXC+1];  /* Y-coordinate of circle */
int   nkoc[MAXC+1];  /* color of circle */
int   numc;          /* number of circles */

int   ncol[MAXL+1];  /* first circle of line */
int   nccol[MAXL+1]; /* second circle of line */
int   nkol[MAXL+1];  /* color of line */
int   numl;          /* number of lines */

int   ic, jc, kc, lc, mc, nc;  /* circle indices */
int   il, jl, kl, ll, ml, nl;  /* line indices */
```

declares the same set in C. The extra element in the declaration is needed since C allocates an extra array element at zero.

A word of warning for the C neophyte: Unlike FORTRAN, the C language has reserved words. When I have a fleet construct (F) in FORTRAN, the following code gives me no trouble:

```
c                npfof(jf) = previous fleet to fleet JF
c                nnfof(jf) = next fleet after fleet JF
      do 510 jf = 1,numf
        if = npfof(jf)
        kf = nnfof(jf)
        call match (if, jf, kf)
510   continue
```

But when I translate that same code into C, the compiler has a tough time since “if” is a reserved word in C. Rather than mess up my pattern of coding, I’ll use `IFF` instead of `IF` for my variable name.

```
               /* npfof(jf) = previous fleet to fleet JF */
               /* nnfof(jf) = next fleet after fleet JF  */
for (jf = 1; jf <= numf; jf++)
{
  iff = npfof(jf);
  kf = nnfof(jf);
  match (iff, jf, kf);
}
```

Lest I appear to take the position that it is perfectly reasonable to use keywords indiscriminantly for variable names, let me show you a piece of code that was shown to me a few years ago (Stone [10]). This horrible example compiles perfectly on almost any compiler that allows a seventh character in variable names and I leave it to you, gentle reader, to figure out what it does!

```
cccccccccccccccccccccccccccccccccccccccccc
c             An example of unambiguous
c             structured FORTRAN programming.
c
      integer go to, do 100 if, return
      character*12 read
      logical go to if
      real logical(30)
      data read / 12h(1x,i5,f8.1) /
      data go to, do 100 if, if 100 do / 2, 1, 30 /
      assign 100 to if go to

      do 100 if = do 100 if,if 100 do
        return = if
        logical(if) = return
        go to if = if-go to*(if/go to).eq.do 100 if
        if (go to if) go to if go to
        call call (logical, return)
100   continue

      print read, (if, logical(if), if = do 100 if,if 100 do)
      stop
      end

      subroutine call (integer, real)
      integer real
      real integer(real)
      common (if) = if*if
      integer(real) = common (real)
      return
      end
```

Just to keep things separate, I generally use a first letter of N for global arrays and a first letter of I for local arrays and function names. I try to use first letters of J, K, and L for scalars.

If a variable doesn't mean anything, then I give it an obviously meaningless name like `ITEMP`, or `FTEMP`. Non-functioning arguments to subroutines get names like `IDUMMY` or `FDUMMY`. Debugging flag variables are `IDEBUG`, `JDEBUG`, and `KDEBUG`.

In FORTRAN, I have parameters for `IFALSE=0` and `ITRUE=1` and, in C, I use #DEFINE statements to set `FALSE` to zero and `TRUE` to one.

If a subroutine or function is short enough, then all of these guidelines can go out the window. Consider STRCPY, a C subroutine taking two pointer to CHAR arguments. It copies a string determined by the second pointer into a string determined by the first pointer. In keeping with my normal guidelines for naming variables, I would code STRCPY as follows.

```
                   /* ***** STRCPY ***** */
void strcpy (qtop,  qsop)
char        *qtop, *qsop;
{
  int jp;
            /*        P = position in string. */
            /* qtop[jp] = target character in position JP */
            /* qsop[jp] = source character in position JP */

  for (jp = 0; qsop[jp] != '\0'; jp++)
    qtop[jp] = qsop[jp];

  qtop[jp] = '\0';
  return;
}
```

But for a routine this small, any programming structure becomes more of an effort to figure out that the subroutine itself! Here I just name the variables so that a casual reader can figure out what the code means without worrying about consistency.

```
    /* ***** STRCPY ***** */
void strcpy (target,  source)
char        *target, *source;
{
  int j;

  for (j = 0; source[j] != '\0'; j++)
    target[j] = source[j];

  target[j] = '\0';
  return;
}
```

We don't have to work hard to remember that `SOURCE` and `TARGET` are string pointers in a five liner like this one. But I only do this if the routine is very short and its function is very obvious.

There is a naming scheme called "Hungarian" notation that has leading characters that tell what things are. While the Hungarian notation specifies a far finer level of detail than I normally use, I find reading programs written this way to be easy and informative.

4.3 Function and Subroutine Names

For functions, I stick to the same implicit type rules I use for variables. For integers, I tend to use J, K, and L for scalers, N and M for arrays, and the letter I for functions. Even though I put a space after a function name and not after an array name, confusion is reduced this way. If I have an array that keeps track of the first line connected to a circle and a function that calculates the last line, the code

```
jl = nfloc(jc)
kl = illoc (jc)
```

makes that difference visible in the spacing and the names.

I try to stick to short names in C (eight characters or fewer) for functions, but have no such reservations about subroutines. So the function to find the last line of a circle might be called ILLOC, the subroutine to generate the array of first lines of circles might be called POPULATE_FIRST_LINE_ARRAY.

If I want to interpret the FORTRAN subroutine definition literally, then I would say a C function is a "subroutine" so long as it doesn't return a value, or "returns a void" in C terminology. I extend subroutines to include functions whose return value is not an essential part of their duty. PRINTF returns an integer, the number of characters printed, but obtaining that integer value is hardly the main reason one uses PRINTF.

Keep the library manual handy in C. Bad things will happen if you step on the wrong library function. I once coded an output subroutine called WRITE and PRINTF stopped working since it called my WRITE subroutine instead of the WRITE subroutine in the library.

4.4 Upper and Lower Case

In FORTRAN, I use lower case for everything except parameters which are entirely upper case. While it is tempting and legal to mix cases, I find it distracting. While no compiler I know has any problem representing the same variable in different case, the ANSI standard requires the same case be used for a symbol every time it appears.

C, on the other hand, is case sensitive. I use lower case for variable names, function names, and subroutine names. I use upper case for #DEFINEd entities such as parameters and macros. I use leading upper case followed by lower case for labels.

```
#define MAXL 1000
#define BIG  1000.0
float fsizoll (int, int), fsiz, fminsiz;
int jl, kl, jlmin, klmin, numl;
 . . .
fminsiz = BIG;  jlmin = klmin = 0;
for (jl = 1; jl <= numl-1; jl++)
{
  for (kl = jl+1; kl <= numl; kl++)
  {
    fsiz = fsizoll (jl, kl);
    if (fsiz < fminsiz)
    {
      if (fsiz == 0)  goto End_Of_Loop:
      fminsiz = fsiz;
      jlmin = jl;  klmin = kl;
    }
  }
}
End_Of_Loop:
printf ("Minimum size is %d and %d\n", jl, kl);
```

5 DATA ORGANIZATION

The only data structure I actually use is the array. I know there are now structured variables in C, but I still don't use them. They tell me that

```
struct circle
{
  float fx;   /* X-coordinate of circle */
  float fy;   /* Y-coordinate of circle */
  float frad; /* radius of circle */
  int   nk;   /* color of circle */
}
oc[MAXC+1];

float fx, fy, frad;

for (jc = 1; jc <= numc; jc++)
{
  find_location (&fx, &fy);
  oc[jc].fx = fx;  oc[jc].fy = fy;  oc[jc].frad = 1.0;
}
```

is supposed to be better than

```
    float   fxoc[MAXC+1];   /* X-coordinate of circle */
    float   fyoc[MAXC+1];   /* Y-coordinate of circle */
    float fradoc[MAXC+1];   /* radius of circle */
    int     nkoc[MAXC+1];   /* color of circle */

    float fx, fy, frad;

    for (jc = 1; jc <= numc; jc++)
    {
      find_location (&fx, &fy);
      fxoc[jc] = fx;  fyoc[jc] = fy;  fradoc[jc] = 1.0;
    }
```

but it just doesn't communicate as well to me.

I use array indices for linked lists as well. Suppose I want to refer repeatedly to all the lines leaving particular circles, that is all `JL` such that `NCOL(JL)` equals `JC`. I would call the first line (on the list) of each circle `NFLOC(JC)` and the next line (on the list) of each line would be `NNLOL(JL)`. For the last line on the list `NNLOL(JL)` would be zero. This code

```
          common nfloc(MAXC)
          common nnlol(MAXL)
           . . .
          do 130 jc = 1,numc
            write (*,100) jc
    100     format (/' Circle =',i4)
            jl = nfloc(jc)
    110     if (jl.eq.0) go to 130
              write (*,120) jl
    120       format (10x,'Line =',i4)
              jl = nnlol(jl)
            go to 110
    130   continue
```

prints the list of lines for each circle. I've seen linked lists done with pointers to structures, but I really have to code it this way before I understand the relationships.

The great advantage of structures over arrays is that a single line of code can copy an entire structure so

```
struct circle
{
  float fx;   /* X-coordinate of circle */
  float fy;   /* Y-coordinate of circle */
  float frad; /* radius of circle */
  int   nk;   /* color of circle */
}
oc[MAXC+1];
 . . .
  oc[jc] = oc[kc];
```

does the same thing as

```
float   fxoc[MAXC+1];   /* X-coordinate of circle */
float   fyoc[MAXC+1];   /* Y-coordinate of circle */
float fradoc[MAXC+1];   /* radius of circle */
int     nkoc[MAXC+1];   /* color of circle */
 . . .
    fxoc[jc] =   fxoc[kc];
    fyoc[jc] =   fyoc[kc];
  fradoc[jc] = fradoc[kc];
    nkoc[jc] =   nkoc[kc];
```

but the original UNIX C compilers did not support this (which in my mind is like selling cars without seats) and the advantage is far outweighed by confusion elsewhere.

I suppose there will be a generation of children who count to five by saying "zero, one, two, three, four." Most computer languages today allow specification of both minimum and maximum indices for an array, but C *requires* the minimum to be zero. I learned to count "one, two, three, . . ." and have become comfortable in C dimensioning one extra element of each array to satisfy my counting habits. My arrays start at one rather than zero whether I program in FORTRAN, C, BASIC, ALGOL, or APL.

5.1 *More on Arrays*

I have been using arrays for thirty-two years. Alternative ways of organizing data have come and gone and I have continued to use arrays. For each primitive (page 10) in my project, I define a set of indexed arrays with its attributes. If each circle has an X-coordinate, a Y-coordinate, a radius, and a display color, then I'll have four arrays indexed from one to the maximum number of circles. I'll keep a scalar integer variable for the actual number of circles the program is currently using.

Looking at those four parallel *arrays gives me a sense of number. For the price of deciding "up front" how many circles I can have, I get a confident sense of size reading the software. If the maximum circle count is one hundred, then I know there won't be tens of thousands of circles.*

Parallel arrays give me control of scope (Section 9.9, page 53). Every function or *subroutine in FORTRAN, every source module in C, tells me right away which arrays it controls. When I find a need to enlarge the scope of one of these arrays, I can move it into another header file with broader access.*

Parallel arrays have inheritance, too. I write software in "layers" where each layer inherits its data architecture from another. An airline scheduling program runs from airports, fleets, and flight legs. A connection builder generates paths (itineraries) and augments the data associated with airports, fleets, and legs while using much of the same source code. But the connection builder adds attributes to airports, fleets, and legs as well and these should merge nicely with the scheduling data. The arrays make it easy to manage scope: the scheduling software only "sees" the arrays it needs and the connection builder sees the broader spectrum of arrays it needs including the schedule arrays.

Array indices are easy to keep track of. When I was growing up, a "linked-list" was a trail of indices. Consider a thousand flight legs assigned to a dozen fleets. It makes sense to have each leg point to the next leg in the same fleet so we can run through just the legs in one particular fleet. We keep an array of leg indices to do this (which I call `NNLFOL`*, iNteger Next Leg by Fleet Of Leg) and we called this array a linked-list of legs by fleet.*

Few things are as easy for debugging purposes as arrays and subscripts. If an array element goes awry, then almost any debugger will let you watch that specific place in memory and will tell you when it changes. And a wayward index is usually easy to spot: if a value is supposed to be a one through twelve fleet index and it comes up as 438239, then, "Houston, we have a problem."

Indexed arrays of one or more dimensions are available in just about all programming languages. When I moved my work from FORTRAN to C, the content of my source code remained the same. (I didn't even have to give up parentheses for brackets using the bounds checking macros described below.) When somebody needs my subroutines to run in Pascal, for example, translation should prove an easy exercise, probably one amenable to a small computer program rather than human transcription. Parallel arrays are close to the way computers actually do things when the source code is compiled.

Finally, arrays should be fast as well as legible. The access to an array element is the array start address plus the index shifted to multiply by a power of two and a memory lookup of the result.

Arrays are the computing equivalent of a sports car. They give me performance and they give me "road feel" and I don't find myself yearning for the Cadillac of alternative data architectures even after a long drive.

The C programming language gives me a nice twist in my parallel array universe: I can dimension my arrays at run time and, with a little memory and disk "hoochy-skooch" I can make them bigger. For the first leg in a fleet, I declare `*NFLOF` *instead of* `NFLOF[MAXF+1]` *and then I allocate memory at run time.*

```
    nflof = (int *) malloc (maxf*sizeof (int))-1;
```

If I don't subtract one at the end, then the array runs from zero to `MAXF-1` *which isn't what* I *want (page 18).*

Arrays of more than one dimension work almost as nicely. Consider the floating point *value of a leg and fleet or the one byte radio radio path gain of cell, X, and Y.*

```
fvolf = (double **) malloc (maxl*sizeof (double *))-1;
for (jl = 1; jl <= maxl; jl++)
  fvolf[jl] = (double *) malloc (maxf*sizeof (double))-1;

ipgocxy = (BYTE ***) malloc (maxc*sizeof (BYTE **))-1;
for (jc = 1; jc <= maxc; jc++)
{
  ipgocxy[jc] = (BYTE **) malloc (maxx*sizeof (BYTE *))-1;
  for (jx = 1; jx <= maxx; jx++)
    ipgocxy[jc][jx] = (BYTE *) malloc (maxy*sizeof (BYTE))-1;
}
```

One this is done, the rest of the program refers to `NVOLF[JL][JF]` *or* `IPGOCXY[JC][JX][JY]` *just as if they were dimensioned at compile time.*

And for those of us who want bounds checking, too, there is another C workaround to this standard FORTRAN feature.

```
#define nflof(jf) nflof[Range(jf,1,numf)]

#define fvolf(jl,jf) fvolf[Range(jl,1,numl)][Range(jf,1,numf)]

#define ipgocxy(jc,jx,jy) ipgocxy[Range(jc,1,numc)]\
                                 [Range(jx,1,numx)]\
                                 [Range(jy,1,numy)]
```

Again we can turn a deficiency into a benefit by setting the array testing against NUMF, the number of fleets we are actually using, rather than just the dimensioned maximum number. So long as we are VERY *careful never to let* `NUMF` *exceed* `MAXF`*, this is a better way to check bounds.*

In the case of events coming and going all the time, as in a simulation, there is a natural parallel array solution as well. When an event is released by the simulation, it goes onto a linked list of available events. (I typically call the first available event `NE_AVAIL` *and the next available event* `NNEOE`*, the same linked-list array I used for the next regular event in the simulation. It makes debugging easier if I also keep* `NE_LAST_AVAIL` *and put the new available events at the* end *of the available list.) At any given point, there is some collection of events between one and* `NUME` *being used by the simulation and a complementary collection available for use, where* `NUME` *stays above the "high water mark" of the simulation.*

Rather than being a kludge, this gives me real control over my events without me having to work hard at it. If some value goes astray, `NNEOE[1843]` *becomes -908493384, then I can run my trusty source code debugger and put a watch on* `NNEOE[1843]` *to alert me when it becomes -908493384. This could happen before event 1843 shows any symptoms.*

Parallel arrays are simple, efficient, direct, and maintainable. These advantages become larger and clearer as program scope and complexity increases. They are not the only *way to manage internal program memory but they are my first choice and they are a viable solution that seems to have been sacrificed at the alter of some post modern programming vision.*

5.2 *More on Data Structures*

Intrigued briefly at first, I have maintained an uncomfortable feeling about data structures. The aesthetic argument is hard to counter as it seems to make so much sense to bundle like things together at a programming language level. Rather than have

```
float   fxoc[MAXC+1];   /* X-coordinate of circle */
float   fyoc[MAXC+1];   /* Y-coordinate of circle */
float fradoc[MAXC+1];   /* radius of circle */
int     nkoc[MAXC+1];   /* color of circle */
```

we would package these four into one "structure" array.

```
struct
{
  float fx;           /* X-coordinate of circle */
  float fy;           /* Y-coordinate of circle */
  float frad;         /* radius of circle */
  int   nk;           /* color of circle */
} oc[NUMC+1];
```

Furthermore, if a program has a few dozen parameters, it is tempting to bundle those together into groups of similar purpose or common usage by putting them into data structures as well. At the expense of a few syllables, the programmer can communicate purpose and context when SPEED and HEIGHT become MOTION.SPEED and POSITION.HEIGHT.

Also, debuggers like data structures. I have written scripts that tell me all the values of each of my parallel arrays, to print `FXOC[7]`*,* `FYOC[7]`*,* `FRADOC[7]`*, and* `NKOC[7]` *but I have to admit it is nicer just to print* `OC[7]` *once.*

It has taken me another decade and change to reaccess that discomfort and realize that I really don't like data structures much. Like a drill bit that makes square holes, they are a specialized tool for a specific purpose and they perform well and communicate much when used properly. And I find them confining, limiting, confusing, and annoying most of the time.

First of all, unless one descends into the realm of object and classes, another realm entirely, they do not allow a program layer gracefully to inherit attributes from another layer. If my main drawing program has the circles described above and I want to add motion, then I add two arrays and I'm done.

```
float fvxoc[MAXC+1];   /* X-velocity of circle */
float fvyoc[MAXC+1];   /* Y-velocity of circle */
```

While some part of me has to recognize the fundamental difference in scope between the circles position (FX,FY) and its velocity (FVX,FVY), the equations of motion need not *be burdened with that kind of program management.*

```
        fxoc[jc] += fvxoc[jc]*ftime;
        fyoc[jc] += fvyoc[jc]*ftime;
```

It's too bad C doesn't have internal complex arithmetic which would have made this so natural.

Now consider the same issue with data structures. We can write an entirely new header file for the motion program that incorporates the drawing attributes

```
    struct
    {
      float  fx;          /* X-coordinate of circle */
      float  fy;          /* Y-coordinate of circle */
      float  frad;        /* radius of circle */
      int    nk;          /* color of circle */
      float fvx;          /* X-velocity of circle */
      float fvy;          /* Y-velocity of circle */
    } oc[NUMC+1];
```

or we could have another data structure for motion

```
    struct
    {
      float fvx;          /* X-velocity of circle */
      float fvy;          /* Y-velocity of circle */
    } draw_oc[NUMC+1];
```

and manage the two data structures in the same dual manner I use for my parallel arrays.

Like landing an airplane in a stiff crosswind, there is no really, truly, wonderfully right way to do it. Inheritance is a real issue and I like my way because it is simplest.

The performance issue may be more subtle. I can't think of any way to implement data structures that doesn't involve an extra integer multiplication. There may be sophisticated optimizations that can reduce that overhead for multiple accesses in the same data structure, but I haven't trusted C optimizers for good reasons. When an optimizer makes my program run faster, that's good; when it makes my program give different answers, that's bad. At best the data structure approach involves extra arithmetic and I have less of an idea what the computer is doing on the inside.

Reading data structure code is usually more difficult, too. The data structure array declarations above aren't so bad, but that's not usually the way it works out. Usually a type definition is in one place and the actual declaration is somewhere else. Throw in a memory pointer or two, more popular than arrays of structures these days, and you have a treasure hunt on your hands to find out what a variable does.

The parameter data structure also makes life more difficult. I'm running along a piece of source code and I see `MOTION.SPEED` *and I know what it* is *but not where it* comes from. *Is this* `MOTION` *structure the same* `MOTION` *structure I saw in some other source file? Again, many programmers heighten the confusion by passing the* `MOTION` *structure, or, worse yet, a memory pointer to it, from subroutine to subroutine so I have* `PMOTION->SPEED` *where*

`PMOTION` *is a function argument. How am I supposed to know that this is the same* `SPEED` *I saw somewhere else?*

One may retort that it's "obvious" that one `SPEED` *is enough. But local manipulation of parameters is perfectly reasonable. I have a graphics editor that has a "symbol" construct which is merely another file (which can have more symbols nested inside it). The parameters of (X, Y) translation, scale, rotation angle, and color are global to this symbol and everything in it, including more symbols, but not to the rest of the program. So it makes sense* in this case *to have parameters of limited scope and it* doesn't *make sense elsewhere.*

The use of data structures for parameters is like introducing a woman as "my current wife" or, worse yet, "one of my wives." It suggests that there have been others or that there may currently be others. I only do that if it is true.

Do I ever use data structures? There are two cases that lend themselves to data structures, one old and one new. The first is memory management, hardly an issue in today's gigabyte world with virtual memory on top of that. In an age where a large machine had 32 kilobytes of random access memory and a large database was several hundred megabytes, keeping the entire file in memory was absurd. As recently as 1988, I was writing software for a 640 kilobyte DOS machine to solve a problem with a few megabytes of internal data and I paged memory back and forth using data structures. When I had the same problem in FORTRAN, I wrote elaborate subroutines to manage "proxy" indices and it was a lot more work. I can think of few examples today where the use of data structures would facilitate management of random access memory.

The second example is a special case use. Sometimes there are a small number of "things" that have a large collection of attributes. Data files for reading and writing are an example, windows on the screen are another. There might be as many as a dozen and they are typically named rather than numbered. Rather than have a dozen or one hundred variables running around for each open file or for each window being viewed, it makes more sense to have a large data structure and to create a new structure for each new file or window. I'm not sure this is really better, but it's the only time I feel comfortable using data structures.

Much of this discussion is aesthetic rather than practical; data structures may inspire other programmers in some way in which I remain unconverted and unseduced. I know I don't like them and I lose no productivity as a result of not liking them.

5.3 *Smart to be Lucky*

There is another issue in data organization that has nothing to do with arrays, data structures, or any of that stuff. It has to do with seeing a picture bigger than the current assignment. So often I'm told "we can't do that" because some parameter of the programming environment is inflexible or unyielding to expansion.

When our group was offering a three-sector product and our competitors were offering a six-sector product, did it really make sense to code the sectors as `FACE0`*,* `FACE1`*, and* `FACE2` *rather than to use an array of sectors? If we came along a year later and decided,* *or if our customers told us that* they *decided, to use six-sector technology, then how are we going to feel about the forty-five person years we invested in* `FACE0`*,* `FACE1`*, and* `FACE2` *code?*

Somehow I seem to be lucky enough that my software expands fairly easily when confronted by unexpected requests for new features. I don't know what the magic is, but I do know that it helps to be thinking about what is over the next metaphorical hill. Never mind that you asked and They told you that They will never need that feature, They may get transferred to another division and the new They may have a different attitude. And my experience tells me that the right attitude will allow a programmer to predict far in advance what the new They will be asking for.

They say it is smarter to be lucky than it is lucky to be smart, and I have had some kind of persistant and repeatable luck in organizing my software to adapt. What I do know is that I have always asked if some arrangement will work on a wide variety of problems and if it will answer a wide variety of questions.

6 *POINTERS AND DATA STRUCTURES*

While I'm loathe to devote an entire chapter of my style document to a style I do not use, memory pointers have become so popular and their problems so widespread that I feel I ought to discuss them at some length. Besides the visible use of pointers and structures in C code, there is the entire object oriented software industry which is a formalization of this style of programming.

An overwhelming majority of young programmers seem comfortable with pointer-structure programming style and they almost universally regard me as a "dinosaur" whose techniques have been obsolete for a long time. As I listen to them talking to each other about memory management, object instantiations, garbage collection, and memory leaks, I recall that we spent the same time talking to each other about faster sorts and searches, more robust data organization, and handling unexpected cases, the memory stuff having been managed by a DIMENSION statement. In the frenzy to manage memory, the whole point of effective programming has been lost.

As a personal example, I inherited a few thousand lines of pointer-structure C code. This was well organized, well written, well documented code with a fat notebook of clear writing explaining how it worked. The programmer was clearly a "pro" who happened to like pointers and structures. He had to be somewhere else and I was hired to finish the job.

The model was supposed to solve a "traveling salesman problem" (TSP) for a network with three extra vertices so the traveling salesman's route would start and end at different places on the printed circuit board. (I spent 1975 Summer as a traveling salesman and I can assure you we never saw anthing like the mathematical TSP algorithm!)

After struggling trying to trace the maze of memory pointers, I went back and reorganized the entire program into parallel arrays. (Lazy as I am, I wrote a program to facilitate this.) And, lo and behold, I found the second extra vertex was missing, only the *first and third were in this program. Given that I knew the problem, I suppose I could go back and verify that the program was one structure memory allocation short, but it was clearer when the last vertex was N+2 instead of N+3.*

As I said above, this programmer was not a slob. He was a careful and meticulous professional doing his best, but the medium of pointers and structures demanded more than that. This was a benign case as there were no stray pointers, no memory leaks, and no lost data.

The passion of a generation of programmers doesn't convince me to join them any more than the passion of 1600 million zealous adherents to one religion converts me. It has convinced me to take the time to examine their premises, however, and I remain unconvinced. The actual body of software produced with this paradigm isn't something I would point to with pride. The burden of proof for a new technology that has had millions of enthusiastic adherents and so little success should be theirs rather than mine, but let me take a few pages to explain my discomfort with their programming paradigm.

6.1 *The New Memory Linked Lists*

While I try to avoid using memory pointers, they have their legitimate uses. The C programming language makes character string manipulation excruciating if one insists on avoiding them and I use them as memory pointers just so long as it takes to make them into arrays. And one uses pointers to return more than one value from a C function. One can also use pointers to functions as a clever way to pass a function as an argument to another function. But that's not what I'm talking about here; I'm talking about the pointer-structure style that has become ubiquitous.

```
typedef struct
{
  double fx;                /* X-coordinate of circle */
  double fy;                /* Y-coordinate of circle */
  double frad;              /* radius of circle */
  double ik;                /* color of circle */
  Circle *next; /* next circle */
} Circle;

struct Circle *pCirc, *pFirstCirc;
     . . .
```

```
pFirstCirc = NULL;
while (read_circle (&fx, &fy, &frad, &ik))
{
  pCirc      = pFirstCirc;
  pFirstCirc = (struct Circ *) malloc (sizeof (Struct Circ));
  pFirstCirc->fx   = fx;
  pFirstCirc->fy   = fy;
  pFirstCirc->frad = frad;
  pFirstCirc->ik   = ik;
  pFirstCirc->next = pCirc;
}
     . . .
for (pCirc = pFirstCirc; pCirc != NULL; pCirc = pCirc->next)
{
     . . .
}
```

6.2 *The GOTOs of Data*

"Pointers are the GOTOs of data." I heard this somewhere and forget where (hence no citation) and it is one of the truest aphorisms I have seen in the programming business. I am actually more tolerant of GOTO statements than most these days, and I have my uses for memory pointers. Here I would like to take that analogy just one step further than it usually goes so that others can see memory pointers as I do.

While we know where a GOTO goes to, we don't know where it came from. Similarly, finding a particular circle `PCIRC` *in the above programming example will usually be difficult. Consider a bug that manifests itself in the one hundred and eighty-third circle's X coordinate. In my array world, I can just watch* `FXOC[183]`*, but here I have* `PCIRC` *set to 1093246745, a perfectly valid memory address of no mnemonic value at all. And it may not be at that same address the next time I run the same program as somebody else may be using that memory. If I am lucky enough to figure out that it is circle 183, then I can use a source code debugger and set* `PCIRC` *to* `PFIRSTCIRC` *and type* `PCIRC=PCIRC->NEXT` *one hundred and eighty-two times to find the problem circle. Barring that sort of tedium, the debugg*or *only can see what the program sees, a seriously limited view of its internal data.*

Let us do the same thing in the code space instead of the data space. We start with a perfectly normal C program written in the sequential style I have been advocating. It prints the first ten "triangular" numbers.

```
#include <stdio.h>
int main ()
{
  int i, j, k, m;

  m = 10;
  for (j = 1; j <= m; j++)
  {
    k = 0;
    for (i = 1; i <= j; i++)
      k += i;

    printf ("\t%d",   j);
    printf ("\t%d\n", k);
  }
  return (0);
}
```

Here is the same program with each source code line labeled.

```
#include <stdio.h>
int main ()
{
  int i, j, k, m;

  M_Init:   m = 10;
  J_Start:  j = 1;
  J_Test:   if (j > m) goto J_Exit;
  Init_Tri:   k = 0;
  I_Start:    i = 1;
  I_Test:     if (i > j) goto I_Exit;
  K_Add:        k += i;
  I_Incr:     i++;                     goto I_Test;
  I_Exit:   printf ("\t%d",   j);
  Print:    printf ("\t%d\n", k);
  J_Incr:   j++;                       goto J_Test;
  J_Exit:   return (0);
}
```

Now let us maintain the sequence but add the extra detail of a "pointer" to the next statement. This should add architectural clarity, should it not?

```
#include <stdio.h>
int main ()
{
  int i, j, k, m;
                                       goto M_Init;
  M_Init:   m = 10;                    goto J_Start;
  J_Start:  j = 1;                     goto J_Test;
  J_Test:   if (j > m) goto J_Exit;    goto Init_Tri;
  Init_Tri:   k = 0;                   goto I_Start;
  I_Start:    i = 1;                   goto I_Test;
  I_Test:     if (i > j) goto I_Exit;  goto K_Add;
  K_Add:        k += i;                goto I_Incr;
  I_Incr:     i++;                     goto I_Test;
  I_Exit:   printf ("\t%d",   j);      goto Print;
  Print:    printf ("\t%d\n", k);      goto J_Incr;
  J_Incr:   j++;                       goto J_Test;
  J_Exit:   return (0);
}
```

As the final stage, let us demonstrate the flexibility of this programming style by showing that the physical order of the program lines is merely a progamming convenience. Only a "dinosaur" should write software dependent on something as transient as statement order.

```
#include <stdio.h>
int main ()
{
  int i, j, k, m;
                                       goto M_Init;
  J_Incr:   j++;                       goto J_Test;
  Init_Tri:   k = 0;                   goto I_Start;
  I_Exit:   printf ("\t%d",   j);      goto Print;
  J_Start:  j = 1;                     goto J_Test;
  K_Add:        k += i;                goto I_Incr;
  J_Exit:   return (0);
  I_Start:    i = 1;                   goto I_Test;
  Print:    printf ("\t%d\n", k);      goto J_Incr;
  J_Test:   if (j > m) goto J_Exit;    goto Init_Tri;
  I_Incr:     i++;                     goto I_Test;
  M_Init:   m = 10;                    goto J_Start;
  I_Test:     if (i > j) goto I_Exit;  goto K_Add;
}
```

Look at the advantages: we can now put the statements in an order dicatated by aesthetics freed from the mundane and confining issues of execution sequence. And if somebody drops the program deck, then all we have to do is find the first few cards and the last few cards *and the program will still run.*

"But wait a minute," I hear you say. "That's STUPID! Nobody would ever write code that way. Stupid, stupid, STUPID!" Well, I don't find that any worse (or any better) than pointer-structure or object oriented programming style. Really, I don't.

6.3 *Objects and Garbage Collection*

I remember when memory management was a bunch of DIMENSION statements and, so armed, we went off to slay dragons. With a little judicious effort, I have been able to use run time memory allocation to advantage. Now I have run time array sizing, a nice feature.

But that's it, the end of the dynamic memory road, at least the end of the part that is paved. As we wander further into the realm of memory management, we trespass onto a rutted dirt road. Even if the programmer doesn't mind the rustic wilderness, his customer may be less pleased to find each run of a program a wilderness adventure.

An operating system allocates and frees memory only so well. When I used IBM AIX, for example, it would only allocate memory in powers of two. Ask for 513 bytes and it used 1024. More painfully, ask for 1048577 bytes and it used 2097152. And my printed circuit board autorouter, aggressively stingy of board real estate, often could not figure out to put a new trace where an old one the same size had been removed. How effective do you think AIX is going to be at filling in its memory holes?

So the program that allocates, frees, and then allocates memory is going to become a memory monster as it runs, spreading until it runs out of random access memory. At best it will act differently on different machines, not a good trait for software to have. And the programmer who forgets to assign a value is going to get inconsistent, non-repeatable behavior, the worst scenario for effective debugging.

And what is the positive side? I have to admit it's more than saving a few kilobytes in a gigabyte world. There is something nice when you don't have to count, when you don't have to decide a priori *how many of something you're going to allow. But when I'm writing my own programs, it isn't worth it.*

6.4 *One Dimensional Thinking*

There is another deeper and more troubling aspect to the pointer-structure or object oriented programming style. We start with a collection of airports daisy-chained together by memory addresses. How do we find the airport "EWR"? Well, we could keep an array of pointers to the structures and sort that array using QSORT or something more robust. But once we have the array, why not use that instead of the house-of-cards of memory-linked pointers?

Chained structures lack the identity of an index, but it's much worse than that. There is no natural way to express any table of two or more dimensions. I think this is a problem in database programming as well, but I'm not as sure as I am right here.

Let's start with a collection of airports daisy-chained and another collection of fleets similarly memory-linked. Now, for each airport and fleet we have some data. There is some minimum time the plane needs between takeoff and landing and that time depends on the airport and the fleet. Maybe there is a requirement as to how many of each fleet have to spend the night at certain maintenance bases. For whatever reason, there is a two dimensional data requirement. And some of my projects have had three and four dimensional data. How are we going to manage that?

I'm sure there is some kind of workaround, but it's a kludge, *a trick that bends the rules to make our data fit into the paradigm. Tables, arrays, and matrices are not an outside irritation to be bent into a programming methodology: These are fundamental parts of a computational problem.*

Over and over again I have seen programmers in the new school retreat from a problem that should be well within their grasp because their techniques don't give them the mental tools to deal with the complexity they have to face. Cowering in the presence of a rat doesn't give me much faith in a self-proclaimed dragon slayer.

There may be a community of new-paradigm programmers out there producing software I would be proud to write. However, in the absence of evidence I'm going to trust my own intuition and experience.

7 INDENTATION

A useful technique for making source code more readable (even for those who use comments), is to indent loop and if-then-else constructs so they are visually distinct. When indentation is insufficient emphasis, then blank lines can be added before and after such a construct. Nested constructs are multiply indented.

I have seen code without any informative spacing and it is truly awful to read. Code with indentation done badly is better, but it really makes a positive difference to pick a reasonable scheme of indentation and stick to it.

I have decided on two spaces for my incremental indent after using one, three, four, and eight spaces. Since I don't like lines over 72 characters on the screen (and neither do some FORTRAN compilers), eight space indentation becomes silly. Two loops with a few if statements starts to become silly with four space indentation. Since my editor makes four and sixteen spaces convenient, I find three space indentation awkward and one is too few to be visually informative.

```
c                One space indent.
      if (ilmax.lt.10) then
       do 140 jl = 1,ilmax-1
        if (nxol(jl).gt.0) then
         do 130 kl = jl+1,ilmax
          if (nxol(kl).gt.nxol(jl)) then
           if (nxol(jl)+nyol(jl).gt.nxol(kl)+nyol(kl)) then
            kountl = kountl+1
           else
            kountl = kountl-1
           end if
          end if
130      continue
        endif
140    continue
      end if
```

```
c                Two space indent.
      if (ilmax.lt.10) then
        do 140 jl = 1,ilmax-1
          if (nxol(jl).gt.0) then
            do 130 kl = jl+1,ilmax
              if (nxol(kl).gt.nxol(jl)) then
                if (nxol(jl)+nyol(jl).gt.nxol(kl)+nyol(kl)) then
                  kountl = kountl+1
                else
                  kountl = kountl-1
                end if
              end if
130         continue
          endif
140     continue
      end if
```

```
c                 Four space indent.
      if (ilmax.lt.10) then
          do 140 jl = 1,ilmax-1
              if (nxol(jl).gt.0) then
                  do 130 kl = jl+1,ilmax
                      if (nxol(kl).gt.nxol(jl)) then
                          if (nxol(jl)+nyol(jl).gt.
     1                                 nxol(kl)+nyol(kl)) then
                              kountl = kountl+1
                          else
                              kountl = kountl-1
                          end if
                      end if
130               continue
              endif
140       continue
      end if
```

7.1 IF-THEN-ELSE

I indent only the contents of the IF section. The ELSE, ELSE IF, and END IF statements are left alone.

```
if (i.eq.1) then
  call first (i)
else if (i.eq.2) then
  call first (i+1)
  call second (i+2)
else
  call last (0)
end if
```

In C, the same construct looks like this.

```
if (i == 1)
  first (i);
else if (i == 2)
{
  first (i+1);
  second (i+2);
}
else
  last (0);
```

7.2 Loops

Similarly, I indent only the contents of a loop. A loop has an unindented beginning and end. In FORTRAN, I use the CONTINUE statement to enforce this. (Furthermore, I use a separate CONTINUE statement for each loop even though FORTRAN allows multiple loops to use the same CONTINUE statement.)

```
c                Move circles 100 units right
c                and 50 units up.
      do 10 jc = 1,numc
        nxoc(jc) = nxoc(jc)+100
        nyoc(jc) = nyoc(jc)+ 50
10    continue
```

In C, I use the closing brace as the loop end.

```
              /* Move circles 100 units right */
              /* and 50 units up. */
for (jc = 1; jc <= numc; jc++)
{
  nxoc[jc] += 100;  nyoc[jc] += 50;
}
```

WHILE loops in C are the same as FOR loops.

```
itest = TRUE;
while (itest)
{
  do_first_stage (nxoc, nyoc);
  do_second_stage (nxoc, nyoc);
  do_third_stage (nxoc, nyoc, &itest);
}
```

Since there is no WHILE statement in FORTRAN, I use IF statements with labels and GOTO statements to accomplish the same thing in a readable manner.

```
      itest = ITRUE
80    if (itest.eq.ITRUE) then
        call stage1 (nxoc, nyoc)
        call stage2 (nxoc, nyoc)
        call stage3 (nxoc, nyoc, itest)
        go to 80
      end if
```

Here is a case where I used a GOTO instead of an IF-THEN construct. Here the GOTO has no obvious merit, but some circumstances will lend themselves toward a "manual" loop with no DO or IF construct and I indent it the same way as I indent a "real" loop.

```
      common nfloc(MAXC)
      common nnlol(MAXL)
       . . .
      do 130 jc = 1,numc
        write (*,100) jc
100     format (/' Circle =',i4)
        jl = nfloc(jc)
110     if (jl.eq.0) go to 130
          write (*,120) jl
120       format (10x,'Line =',i4)
          jl = nnlol(jl)
        go to 110
130   continue
```

7.3 C Braces

There are four reasonable ways to align braces.

```
                /* 1.  Unindented aligned braces. */
j_positive = 0;
for (jc = 1; jc <= numc; jc++)
{
  if (nxoc[jc] >= 0 && nyoc[jc] >= 0)  j_positive++;
}
                /* 2.  Unindented unaligned braces. */
j_positive = 0;
for (jc = 1; jc <= numc; jc++) {
  if (nxoc[jc] >= 0 && nyoc[jc] >= 0)  j_positive++;
}
                /* 3.  Indented aligned braces. */
j_positive = 0;
for (jc = 1; jc <= numc; jc++)
  {
  if (nxoc[jc] >= 0 && nyoc[jc] >= 0)  j_positive++;
  }
                /* 4.  Indented unaligned braces. */
j_positive = 0;
for (jc = 1; jc <= numc; jc++) {
  if (nxoc[jc] >= 0 && nyoc[jc] >= 0)  j_positive++;
  }
```

Any of these is reasonable enough so long as one uses the same pattern throughout a program.

Having said that, I have a preference for the first case, unindented matched braces.

I prefer the braces to be part of the unindented code and like the closing brace to be a visible loop ending. Furthermore, having the opening brace and closing brace lined up on the page causes me fewer errors.

Of course, if the code between the braces is short enough and clarity is improved, then I put them on the same line.

```
#define LEFT        1
#define RIGHT       2
#define UP          3
#define DOWN        4
#define UP_LEFT     5
#define UP_RIGHT    6
#define DOWN_LEFT   7
#define DOWN_RIGHT  8
 . . .
for (jc = 1; jc <= numc; jc++)
{
  m = movement (jc);
  if      (m == LEFT)        { nxoc[jc]--;             }
  else if (m == RIGHT)       { nxoc[jc]++;             }
  else if (m == UP)          {             nyoc[jc]++; }
  else if (m == DOWN)        {             nyoc[jc]--; }
  else if (m == UP_LEFT)     { nxoc[jc]--; nyoc[jc]++; }
  else if (m == UP_RIGHT)    { nxoc[jc]++; nyoc[jc]++; }
  else if (m == DOWN_LEFT)   { nxoc[jc]--; nyoc[jc]--; }
  else if (m == DOWN_RIGHT)  { nxoc[jc]++; nyoc[jc]--; }
}
```

This is where C really shines. No construct in FORTRAN says this as well and no combination of SWITCHes and CASEs does either. There may be some bit twiddling combination using `&` and `|` with the right numbers for LEFT, RIGHT, *et cetera* that gets the whole movement into a one line expression and runs 0.03 percent faster, but I find this code crystal clear without a single comment!

7.4 C SWITCH Statements

Now that I have said I seldom use SWITCH statements in C, I'll tell you how I indent them. I indent SWITCH statements twice, once for the SWITCH and once for the CASE.

```
                /* A does ALPHA          */
                /* B does BETA           */
                /* C does ALPHA and BETA */
                /* D does ALPHA and BETA */
    switch (qinput)
    {
      case 'a':
        do_alpha ();
        break;
      case 'b':
        do_beta ();
        break;
      case 'c':  case 'd':
        do_alpha ();
        do_beta ();
        break;
    }
```

While we are on the subject, the SWITCH statement in C is totally non-structured in a terrible way. If you forget the BREAK statement, the computer runs right on into the next case! This supposedly adds "flexibility" to C without having to use a GOTO statement. Instead, it creates a construct where a less informative BREAK statement performs the exact same function. While the following code is legal, you'll never figure out what you did six months from now.

```
                /* A does ALPHA          */
                /* B does BETA           */
                /* C does ALPHA and BETA */
                /* D does ALPHA and BETA */
    switch (qinput)
    {
      case 'a':  case 'c':  case 'd':
        do_alpha ();
        if (qinput == 'a') break;
      case 'b':
        do_beta ();
        break;
    }
```

7.5 C Declarations

I indent declarations in C the same way I indent code. When I declare structures, I do it my usual way. Compile time array assignments are done it whatever way communicates best. In subroutines, I align the declarations of arguments up with the arguments.

```
#include <stdio.h>
int ic, jc, kc, lc;
int il, jl, kl, ll;
int nsizoc[5]  = { 0, 14, 12, 10,  8 };
int nsizol[31] = { 0, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39,
                      38, 37, 36, 35, 34, 33, 32, 31, 30, 29,
                      28, 27, 26, 25, 24, 23, 22, 21, 20, 19
                 };
struct order
{
  float x;  /* X-coordinate */
  float y;  /* Y-coordinate */
  float z;  /* Z-coordinate */
  int   k;  /* color */
} a, b;
void align (int, float *, int, float *, int, float *);

                /* ***** MAIN ***** */
main (argc, argv)
int   argc;
char      **argv;
{
  int l, m, n;
  float u, v, w;
   . . .
  align (l, &u, m, &v, n, &w);    /* float pointers */
   . . .
}

                /* ***** ALIGN ***** */
void align (i1, p1, i2, p2, i3, p3)
int         i1,     i2,     i3;
float          *p1,    *p2,    *p3;
{
  float a1, a2, a3;
  a1 = *p1;  a2 = *p2;  a3 = *p3;   /* float values */
   . . .
  return;
}
```

8 COMMENTS

The most important comment in any programming language is a blank line. (Since the official FORTRAN does not allow blank lines, I use a blank comment instead with a lone C in column one.) I use a blank comment between loops, between groups of lines, *et cetera.* After the header comment, the spacing between code is enough to make it perfectly clear (to me) what is happening.

```
c                Sort lines in left-to-right
c                order in their left endpoints.
      do 220 jl = 1,numl-1
        jc1 = ncol(jl)
        jx1 = nxol(jc1)
        jc2 = nccol(jl)
        jx2 = nxol(jc2)
        jx  = min (jx1, jx2)
c
        do 210 kl = jl+1,numl
          kc1 = ncol(kl)
          kx1 = nxol(kc1)
          kc2 = nccol(kl)
          kx2 = nxol(kc2)
          kx  = min (kx1, kx2)
c
          if (jx.le.kx) go to 210
c
          itemp    = ncol(jl)
          ncol(jl) = ncol(kl)
          ncol(kl) = itemp
c
          itemp     = nccol(jl)
          nccol(jl) = nccol(kl)
          nccol(kl) = itemp
c
          jx = kx
210     continue
220   continue
```

After using two tabs for a long time, I now indent comments three tabs in FORTRAN. I find the extra indent removes ambiguity between code and comment. Since code in C starts one tab to the left of code in FORTRAN, I only indent C comments two tabs.

I also write more or less grammatical comments with a capital letter at each sentence beginning and a period at the end. When there is something to explain, I put in a paragraph of explanation with blank lines of separation. Many times I've made diagrams in my comments including hyphens, vertical lines, slashes, backslashes, and greater than and less than signs used as arrowheads.

```
c          This section of code adds a new circle
c          to the linked list structure.  We break
c          the link between circles IC and KC to
c          insert a new circle JC.  The new links
c          are IC to JC and JC to KC.
c
c           . . . IC -- remove this link --> KC . . .
c                     \                   /
c        add link here \                 / and here
c                         ----> JC ---->
c
c          The array NFCOL points to the first circle
c          on the linked list for a line and the array
c          NNCOC links each circle to the next circle
c          on the list.
```

When I am adding a feature to this code two years from now, I'll be glad I explained it in detail.

I put a header with stars at the beginning of any subroutine. Where useful, I frame a paragraph telling me what it does. In FORTRAN, I put a comment between subroutines with just a CONTROL-L (page feed) so that the printer starts each new subroutine on a new page.

```
       . . .
      return
      end
c^L
c***************************************************
      subroutine sortc
c***************************************************
c     SORTC sorts circles in left to right order. *
c     If the circles have the same X-coordinate,  *
c     then SORTC sorts them bottom to top.        *
c***************************************************
      include 'circline.hdr'
       . . .
```

In C, I highlight and capitalize the subroutine name. This way I can use a case sensitive search for "SORTC" to find the subroutine itself.

```
                /* ***** SORTC ***** */
void sortc ()
{
            /* SORTC sorts circles in left to right order. */
            /* If the circles have the same X-coordinate,  */
            /* then SORTC sorts them bottom to top.        */
   . . .
}
```

9 PROGRAMMING STYLE

9.1 An Excerpt from a Wonderful Book

I copied this section from a delightful book (Simon [9]) called *Why You Lose at Bridge*. I changed the bridge jargon to software terminology, but the tone and the message are unchanged. If you play bridge, then I highly recommend you read this book. You'll be a better programmer for it!

"As you may have inferred I hold some very positive views about software. And while I am in no way concerned with swaying you from the programming environment you happen to favor, this does not seem to me to be a reason for concealing my contempt for some of them. If at the end of it you are converted from programming in some particular brand of scientific nonsense imbibed from experts who ought to know better, I'm delighted. If not – that's fine too. I only hope I am rated against you at raise time.

"And so I'm going to open my discussion with a warning against that super-scientific but insidious poison that is oozing out of a group of perverted experts to infect masses of well-intentioned, eager-to-learn Computer Science students – turning them from honest straightforward programmers that it is a pleasure to work with into muddle-headed idiots lost in a nightmare of undigested, misapplied, and, in the main, unsound theory.

"Semicolon languages with recursive subroutine calls! Complicated, 'object-oriented' data structures! GOTO-less, 'structured' programming! An operating system with no file, disk, or memory management! Its miserable corollary, source code control!

"It is possible that the poison has not reached your circle. It will. When it does, ignore the convincing scientific patter that accompanies it. Study the results the infected software developers achieve. That's all I ask. STUDY THEIR RESULTS!

"Let there be no confusion. I am not running down scientific programming. I am running down super-scientific gibberish.

"Some programming jobs demand delicate treatment, others are best dealt with poppa-momma fashion. Reserve your science for the jobs that need them. Do not get scientific on every job merely for the sake of science.

"For that is just what the super-scientists do. They are striving for an accuracy of programming that does not exist in software and, in striving for it, they turn the simplest assignments into problems. They have to code twice as well to achieve the same results as a normal programmer. And they don't achieve it.

"I shall always remember with pleasure an inquiry addressed to me by some of these super-scientific software weenies. How, they inquired, had I managed to code a complicated radio control algorithm in a short time. They used sophisticated source code control and multiple levels of code review among fifteen programmers to have the job run on for three years while they struggled to keep the program within a one Megabyte memory limit.

"The subroutine I had written was one thousand lines of code and took up sixteen Kilobytes of memory. I didn't use data structures, object oriented or otherwise. My program was not recursive and I used GOTO statements. It was liberally commented. And when somebody else needed to make a change in it, he only needed fifteen minutes of my time to understand what was in it.

"If these scientists kept their ideas to themselves, it would not be so bad. At least they understand what they are trying to do and why. But they WILL teach others.

"It may be all right for the super-scientist to use recursive programming to do a tree-search. I am liable to do it myself if the mood seizes me. But if I do it, I am experimenting and am fully prepared to apologize profusely if it takes days to find most of the bugs in the recursive code.

"But the super-scientist will teach the eager student that recursion is the ONLY correct way to search through trees and that managing the tree structure in the subroutine code is out of order. If you program the structure explicitly, he explains patiently, you will have to use arrays and pointer variables to refer to elements of the tree. And the software novice listens agog to find himself a few projects later debugging code nested seven levels deep in recursion following arrays and pointer variables that do not match the stack variables.

"Advocates of recursive programming will please refrain from writing to point out that the whole point of recursive programming is not to have arrays and pointers to keep track of previous levels, so such a programming nightmare is impossible. Because it still seems to happen.

"Such teaching is not merely bad software design. It's cruelty to programmers.

"Let me implore you, for the benefit of your overtime and your management's deadlines, to keep your programming as simple as you can. Aim at using the minimum rather than the maximum number of programming constructions to solve your problem. The fewer the constructions the fewer the opportunities to make mistakes. Follow the principle laid down by Baccus and Hopper, adopted gratefully by me, and jeered at as dull and uninspired by scientific and scintillating software weenies, who invariably end up years late in any software project:

WRITE A PROGRAM THAT SOLVES THE PROBLEM.

This still permits you to be as scientific as is necessary, on the jobs that demand it. You can structure, recurse, yacc, lex, while, do until, break, continue, and produce the whole bag of tricks as it is required. But not when it isn't. The moment you know how to solve the problem in front of you, write the program without any further nonsense.

"Keep your programs simple. Use clever tricks when you must and take the direct route whenever you can. Never feel compelled to use a convention where it cannot help you, merely because you happen to know it. If you know you want to get to another part of the algorithm, use a GOTO statement. Don't bother formulating a complicated IF-THEN-ELSE construction. Why make life more difficult for the engineer who inherits your code and has to figure out what you were thinking?

"Whether you are coding an operating system call, a file utility, or a complex mathematical problem, the moment you have decided how the problem should be solved – write the program. It may not be very spectacular, it may evoke the jeers of those who like to wander round the world to reach a result that can be reached in a few lines of code; but, believe me, it pays. You will waste many fewer hours during the year if you stop trying to be 'scientific' all the time.

"That, at least, is my emphatic opinion. You can accept it, investigate it, or ignore it for it is only an opinion; and there are many who disagree with it."

9.2 The GOTO Statement

Yes, I use GOTO statements. I don't use them when an IF-THEN-ELSE construct or a DO loop does the job, but there are too many cases where they don't quite hack it. Most of my GOTO statements in FORTRAN perform the function of BREAK and CONTINUE statements in C.

```
    do 40 jc = 1,numc
      if (nxoc(jc).le.0) go to 40
      if (nyoc(jc).le.0) go to 40
      if (nkoc(jc).ne.1) go to 50
       . . .
40  continue
50   . . .
```

```
for (jc = 1; jc <= numc; jc++)
{
  if (nxoc[jc] <= 0)  continue;
  if (nxoc[jc] <= 0)  continue;
  if (nkoc[jc] != 1)  break;
}
 . . .
```

There is a group of hard core fanatics who don't use CONTINUE statements in C since they violate structured programming. They say that we should use a sequence of nested IF statements instead.

```
for (jc = 1; jc <= numc; jc++)
{
  if (nxoc[jc] >= 0)
  {
    if (nyoc[jc] >= 0)
    {
      if (nkoc[jc] != 1)  break;
       . . .
    }
  }
}
```

When there are a lot of criteria being filtered against, this can result in five or six indentations and code that is hard for me to read. I tend to comment my filters (the first time, anyway).

```
c                For each color JK . . .
      do 330 jk = 1,numk
c                We are setting NNCOK(JK) to the
c                number of circles of color JK
c                that contain the origin (0,0).
        nncok(jk) = 0
c                For each circle JC . . .
        do 320 jc = 1,numc
c                Are we the right color?
          if (nkoc(jc).ne.jk) go to 320
c                FRAD is the radius of the circle?
          frad = fradoc(jc)
c                Are we within range in the X direction?
          if (abs (fxoc(jc)).gt.frad) go to 320
c                Are we within range in the Y direction?
          if (abs (fyoc(jc)).gt.frad) go to 320
c                How far (squared) are we from the origin?
          fx = fxoc(jc)
          fy = fyoc(jc)
          dist2 = (fx*fx)+(fy*fy)
c                FRAD2 is radius squared.
          frad2 = frad*frad
c                Are we further from (0,0) than radius?
          if (dist2.gt.frad2) go to 320
c                Increment number of circles of
c                color JK containing (0,0).
          nncok(jk) = nncok(jk)+1
320     continue
330   continue
```

Other GOTO statements turn FORTRAN IF blocks into WHILE loops.

```
      jc = nfcol(jl)         jc = nfcol[jl];
230   if (jc.ne.0) then      while (jc != 0)
         . . .               {
        jc = nncoc(jc)           . . .
        go to 230              jc = nncoc[jc];
      end if                 }
```

Even in C I use an occasional GOTO statement. Getting out of a double loop is a typical example where I use a GOTO.

```
for (jl = 1; jl <= numl-1; jl++)
{
  for (kl = jl+1; kl <= numl; kl++)
  {
     . . .
    if (itestoll (jl, kl))  go to End_Of_Loops;
     . . .
  }
}
End_Of_Loops:
```

There are those who set a flag variable on the inner loop and BREAK to the outer loop which tests that variable and BREAKs out when the flag is set. I find that an awkward construct and proof that for these programmers avoiding the GOTO is more important than creating readable code.

```
iflag = FALSE;
for (jl = 1; jl <= numl-1; jl++)
{
  for (kl = jl+1; kl <= numl; kl++)
  {
     . . .
    if (itestoll (jl, kl))
    {
      iflag = TRUE;
      break;
    }
     . . .
  }
  if (iflag)  break;
}
```

I don't create IF-THEN constructs just to avoid a GOTO. If there is an entire section of code that I run when a flag is set, I'll often just use a commented GOTO to get there.

9.3 *The C COMMA Operator*

On page 35 we had an example of a C program with so little between braces that it made sense to put the entire construction on one line. The C comma operator is simple enough: it evaluates the first part and then the second part. This is the same program using the comma operator, a perfect example where it improves communication.

```
#define LEFT        1
#define RIGHT       2
#define UP          3
#define DOWN        4
#define UP_LEFT     5
#define UP_RIGHT    6
#define DOWN_LEFT   7
#define DOWN_RIGHT  8
 . . .
for (jc = 1; jc <= numc; jc++)
{
  m = movement (jc);
  if      (m == LEFT)        nxoc[jc]--;
  else if (m == RIGHT)       nxoc[jc]++;
  else if (m == UP)                      nyoc[jc]++;
  else if (m == DOWN)                    nyoc[jc]--;
  else if (m == UP_LEFT)     nxoc[jc]--, nyoc[jc]++;
  else if (m == UP_RIGHT)    nxoc[jc]++, nyoc[jc]++;
  else if (m == DOWN_LEFT)   nxoc[jc]--, nyoc[jc]--;
  else if (m == DOWN_RIGHT)  nxoc[jc]++, nyoc[jc]--;
}
```

I find many opportunities to save space and improve communication using the comma operator. It makes "one line" statements that say what they mean.

```
    char q[128], r[128];

    gets (q);

                  /* Strip trailing space from Q. */
    j = strlen (q)-1;
    while ((j >= 0) && (isspace (q[j]))  q[j] = '\0', j--;

                  /* Print each space delimited string. */
    j = 0;
    while (TRUE)
    {
      i = 0;
      while (isspace (q[j]))                    j++;
      while (isgraph (q[j]))  r[i] = q[j], i++, j++;
                              r[i] = '\0';
      if (i == 0)  break;

      printf ("Next string is %s.\n", r);
    }
```

I tend not to use the comma operator in FOR statements which is where C textbooks put it, at least those I have seen.

```
                  /* Remove spaces from string Q. */
  for (j = 0, k = 0; q[j] != '\0'; j++)
    if (isgraph (q[j])
      q[k] = q[j], k++;

  q[k] = '\0';
```

I put the K=0 statement before *the loop to make it visibly separate from the iteration process.*

The comma operator removes the last excuse to use embedded increment (`++`*) and decrement (*`--`*) operators like* `R[I++]=Q[J++]` *when* `R[I]=Q[J],I++,J++` *says the same thing so much better. I put embedded increments and decrements in the category of cute obfuscation rather than serious programming. Consider these two statements the first of which appears on page 50.*

```
        while (*target++ = *source++);

        while (*target = *source, target++, source++);
```

Never mind what they do or why somebody wants to do it, the second makes the operations apparent while the first is an exercise in head scratching.

9.4 Integer and Logical Variables in C

In FORTRAN there are LOGICAL variables that take the values .TRUE. and .FALSE.. I almost never use them just because I am satisfied to set an INTEGER variable to ITRUE (one) or IFALSE (zero). In C, on the other hand, there are no specific variables for true and false; rather we have arithmetic comparisons returning one for true and zero for false. The statement

```
    printf ("The value is %d\n", (a == b));
```

is perfectly legal and meaningful. Also, IF statements have an implied not-equal-to-zero so the two statements

```
    if (itest != 0)
    if (itest)
```

mean the exact same thing. I am casual about using these two interchangeably. As a general rule, I specify the zero only if it is a numeric value rather than a falsehood or string terminator. Since a bang (!) means logical negation, the two statements

```
    if (itest == 0)
    if (!itest)
```

mean the exact same thing. I used to use the second form regularly and I stopped when I realized that I often read code incorrectly because I failed to see the bang. Instead I use

```
    if (itest == FALSE)
```

to make my meaning clear (with FALSE set to zero).

I am no longer casual about the distinction between the two statements

```
    if (itest != 0)
    if (itest)
```

in my programming. When I read my code later, the first statement tells me that ITEST can take on many integer values including zero while the second tells me that ITEST is either TRUE or FALSE. This is the distinction in FORTRAN between integer and logical variables.

9.5 Pointer Arithmetic in C

Pointers have been hailed as one of the great breakthroughs of C. When I need them, I tend to agree with those folks. But when it comes to writing confusing code, few techniques can beat complex pointer notation. Let's take our old standby, the STRCPY subroutine to copy a source string into a target.

```
                    /* ***** STRCPY ***** */
    void strcpy (target,  source)
    char        *target, *source;
    {
      int j;
      for (j = 0; source[j] != '\0'; j++)
        target[j] = source[j];
      target[j] = '\0';
      return;
    }
```

What a nice piece of code! Without a single comment, I can read this code and visualize my index finger J running along the SOURCE string copying it into the TARGET string until it hits a null termination.

Let us make a simple notational change. Instead of using an index J, let us use CHAR pointers P and Q.

```
                    /* ***** STRCPY ***** */
    void strcpy (target,  source)
    char        *target, *source;
    {
      char *p, *q;
      for (p = source, q = target; *p; p++, q++)
        *q = *p;
      *q = '\0';
      return;
    }
```

Gone is much of the clarity. Rather than daintily running my index finger along the string, now I am banging along it with my thumb. But the pointer zealots aren't through yet. They are quick to point out that the whole FOR loop is unnecessary and can be replaced with a simple WHILE. Furthermore, since the pointers SOURCE and TARGET are already on the stack, we can use them instead of new variables P and Q. (You can keep pointer notation and the stack clear in your mind as you browse through old code, can't you?)

```
                /* ***** STRCPY ***** */
void strcpy (target,  source)
char        *target, *source;
{
  while (*source)
  {
    *target = *source;
    target++; source++;
  }
  *target = '\0';
  return;
}
```

And the cost in clarity, we are told, is minor and the convenience is great if we fully utilize the power of assignments within WHILE statements and the rules for incrementing after other operations.

```
                /* ***** STRCPY ***** */
void strcpy (target,  source)
char        *target, *source;
{
  while (*target++ = *source++);
  return;
}
```

By now, our pretty subroutine has become an awful mess. Yes, I have seen this mess touted as an example of the beauty of C programming and those touting such should be kept away from computers. Good, clean, readable C can be written and it is more likely to be written if people try to stay away from pointer arithmetic.

Let me give another example. If I am using PRINTF, it is easy to concatenate output.

```
                printf ("A=%f ",  a);
if (j_print_b)  printf ("B=%f ",  b);
                printf ("C=%f\n", c);
```

Suppose we want the same output using SPRINTF into a string.

```
char q[100];
int j;
 . . .
                j  = sprintf (q,   "A=%f ", a);
if (j_print_b)  j += sprintf (q+j, "B=%f ", b);
                     sprintf (q+j, "C=%f",  c);
```

Since PRINTF, FPRINT, and SPRINTF return the number of characters they print, the index J counts how far into the Q string the next segment should be. The less clear way of writing the same code in fewer characters uses pointer arithmetic.

```
char *p, q[100];
 . . .
                 p  = q+sprintf (q, "A=%f ", a);
if (j_print_b)  p +=   sprintf (p, "B=%f ", b);
                       sprintf (p, "C=%f",  c);
```

I suppose there are folks who find the second case as clear as the first, but I am not one of them. I have to sit there and count on my fingers to make sure that the CHAR pointer P really points where I want it to point.

9.6 *Loops*

The C language allows a great deal of flexibility in its loop structure at the price of numerical efficiency. The other price is a lack of internal structure. The reader of a program is never quite sure what varies and what stays constant.

I am comfortable occasionally incrementing or decrementing the loop index or boundary inside a C loop, but I try not to make a habit of it.

```
for (jc = 1; jc <= numc; jc++)
{
  if (i_keep_this_circle (jc))  continue;
            /* To delete circle, copy last circle NUMC */
            /* to JC and reduce the list size by one. */
  circle_copy (jc, numc);
  jc--, numc--;
}
```

Another place where C allows me some flexibility is being able to put linked lists in FOR loops. If `NFLOC` *is the first line of the circle and* `NNLCOL` *is the next line with the same circle, then I'm happy with a loop that runs through all of a circle's lines.*

```
for (jl = nfloc(jc); jl != 0; jl = nnlcol(jl))
{
  . . .
}
```

Other than these excursions, I stick with FORTRAN loops. I find that there is a certain comfort in knowing what kinds of loops I'm going to see in my code when I come back to it a few years later.

9.7 Global Variables vs. Arguments

I've written code in BASIC without separate functions or subroutines, only GOSUB statements and all variables were global. I've written code in FORTRAN with no global variables at all where everything a subroutine needed was passed in a long list. Now I have drifted back toward the old way. I only pass information as arguments that is specific to the subroutine.

In FORTRAN, I use the almost standard INCLUDE statement to have a giant COMMON block with all my arrays, important constants, and operating parameters. In C, things are a little dicey since all but one of the declarations of an array have to have the keyword EXTERN. The trick is to #DEFINE a variable MAIN in the main program to be one and elsewhere to be zero and have the header file start with

```
#if MAIN
  #define EXTERN
#else
  #define EXTERN extern
#endif
```

and use the capitalized EXTERN before all variable declarations.

Remember that the FORTRAN world ends at the END statement and the header file must be included in each subroutine or function. In C, each source file should have the header file included before any of its programs, subroutines, or functions begin. *We will have more to say about global variables in Section 9.9 (page 53).*

9.8 Speed vs. Readability

It is my experience that code can get pretty fast before it must be hard to read. It is certainly possible to make very slow code hard to read, but with effort one can make clear to the reader all but the very fastest code.

There comes a point in every programmer's life, however, when there is a clear fork in the road. Nine times out of ten, I'll vote on the side of clarity. Making it fast just isn't worth the extra debugging pain and the far greater pain of trying to figure out later what this code meant today. But when a piece of code runs sufficiently often (on the order of millions of times) that I hate waiting for it to finish, then I'll make clarity compromises necessary to make it run faster.

Where does one start compromising? Or more importantly, where do I look for compromise? First, I look at sorts and searches. Are there swap-sorts or bubble-sorts that I wrote knowing I could fix them later? Usually they can be replaced by heapsorts or non-recursively coded quicksorts. Are there places where I search an entire list for candidates because I never realized it could get this long? A search with a filter often can be turned into a linked list. Searches to describe a particular element can be ordered and binary chopped. Lists with complicated reads and writes updating it all the time can be hashed.

After I've answered the basic issue of minimizing how often code has to run, now I can address how fast it should be. Between zero and one, the difference between $tan^{-1}x$ and $\frac{x}{1+0.28x^2}$ is half a degree or less. When I was calculating radiation patterns of an antenna, for example, then this was more than adequate and at least ten times faster. It is just as accurate to compare x^2+y^2 to r^2 as to compare $\sqrt{x^2+y^2}$ to r and a multiply is oodles of times faster than a square root. It is more subtle when the square root is hidden in the complex absolute value function CABS, but just as costly.

The subject of numerical considerations in programming could fill several good books and has done so. The Handbook of Mathematical Functions, AMS 55 [5] is an invaluable source of fast and good approximations. Listings of many useful subroutines to do simple things I never have the time to figure out myself can be found in Numerical Recipes (Press). For an understanding of how numerical instabilities arise and what to do about them, I heartily recommend Forman Acton's red book [6].

Another alternative for hairy floating point mathematical functions is a table lookup, usually using interpolation. It usually isn't too hard to generate the table and the code is straightforward.

```
function f(x)
dimension table(0:100)
data table / . . . /
if (x.lt.0.0.or.x.ge.1.0) then
  write (*,*) 'Range error in F.'
end if
total = x*100.0
ipart = ifix (total)
fract = total-float (ipart)
f = ((1.0-fract)*table(ipart))+((fract)*table(ipart+1))
return
end
```

Don't be too shy about writing your own substitutes for library functions, but *always* document in comments any limitations your function has that the library function does not have! These limitations can be domain limitations or accuracy limitations. Sometimes you have to write your own function when the libary function just doesn't work. DOUBLE PRECISION COMPLEX library functions on the third-quadrant often have interesting properties. One time I had a library that used the trigonometric functions on the Intel 80387 chip and I was running the code on a machine that only had an Intel 80287 coprocessor. I rewrote SIN and COS to solve that problem.

Last on my list is writing code to make the compiler more efficient. FORTRAN arrays run slightly faster on some compilers starting from zero. Some C compilers create faster executable code from pointer arithmetic code than from code with indices. I've seen C compilers that compare four byte INTs for equality many times faster than they compare the same four bytes for equality when they are FLOATs. These are tricks that wed code performance to a particular environment and that make code hard to read. Sooner or later, if you need it bad enough, then you will have it bad enough.

9.9 *Two Kinds of Scope*

A computer program communicates with itself by storing values in variables. Controlling the flow of a program's internal information is important and, therefore, controlling the scope of variables is important. FORTRAN gave us named COMMON blocks to do this.

```
      program myprog
c                 Nodes and Arcs.
      paramater (MAXA = 180)
      paramater (MAXN = 100)
      common /search/ numn, numa, neon(MAXN), neoa(MAXA)
c                 System stuff, routes and locations.
      parameter (MAXR = 1500)
      parameter (MAXL =  500)
      common /system/ numr, numl, nlor(MAXR), nllor(MAXR)
              . . .
```

The "inner algorithm code" that deals with the graph theory, nodes and arcs, doesn't have to share its variables with the rest of the program that deals with routes and locations. So we name the node-arc stuff SEARCH and the the regular route-location stuff SYSTEM. We can put these in header files that are included in each relevant subroutine using the all-but-standard INCLUDE statement.

In addition to the clarity of communication, there is another advantage to the control of variable scope. *Somebody may reuse a variable name, not always on purpose, and it makes more sense not to have to worry about it. Even a programmer flying solo is sharing the work with libraries and* they *have variables, too.*

The C programming language doesn't have this control. Variables are controlled "geographically" rather than by name, a big step backward. A variable can be local to a function, like FORTRAN, or it can be global to everything or it can be global to a single source file by declaring it outside a function and calling it STATIC. Or it can be declared in the middle *of a source file to be global for none of the above and all of the below. (Haven't I seen that on a multiple choice exam somewhere?) Anything else I want to control has to be done using tricks that make a program hard to follow.*

That means the C programmer has to be concerned with global names throughout his program as he has no way to sequester any variable used in two source files. Back in the early days, it was not unusual for those writing libaries to name their global variables PAGE or STEP or SCAN and the programmer would stub his toe on these without warning. Often it took a long time to find these, especially if the library source was not available.

My convention of short variable names and a single letter for each primitive makes me want more scope control than C allows gracefully. When I want a limited-scope primitive (page 10), say a list of events that I don't want to manage as a linked-list, then I have to put all its relevant functions in one source file even if that is awkward for other reasons. The alternative is having that list of events be global everywhere and losing all those variables names for any other use elsewhere.

But there is another notion of scope that seems to have been lost in the shuffle. To distinguish it from the other kind, let me call this Type Two scope: How many places to I have to look to find out what some variable is?

Consider a reference in PIXMAP.C to

```
                pWin->PixelUp(x,y)
```

which isn't a particularly terrible expression. Well, we scroll around a bit and find that X and Y are local integers and, given the other names, they're probably some pixel location on a window panel. Okay, what about `PIXELUP` *and* `PWIN`*? Well, we root around through PIXMAP.C and its six included header files (other than the standard library headers) and in WINPIX.H we find an included file called SPACE.H and that has a big clue.*

```
                WinSpace pWin;
```

Okay, what is WINSPACE? Well, in WHEADER.H is included WDEFS.H which has the type definition we're looking for.

```
                typedef struct
                {
                  double dMapX;
                  double dMapY;
                  double dTheta;
                  SpaceF PixelUp;
                  SpaceF PixelDn;
                } WinSpace;
```

It doesn't take too much longer to find, in FCNDEFS.H, what SPACEF is.

```
                #define SpaceF PixelConstruct
```

Okay, we go back to WHEADER.H which has the final answer.

```
                typedef void (*PixelConstruct)(int x, int y);
```

We just found out what *we're looking for. We still have to find out which* `PIXELCONSTRUCT` *is attached to* `PWIN` *whose pedigree we have to track down. What was it we were looking for again?*

Pascal programmers have another twist. They can use the WITH statement to add another level of misdirection by putting the structure name far away from the component.

```
                with acirc
                begin
                 . . .
                    if size > 1
                 . . .
                end
```

You look for SIZE and it is nowhere to be found. You look all around and there is no sign of SIZE. But somewhere in a header file far, far away is a data structure declaration of `ACIRC` *as a CIRCLE structure with a* `SIZE` *component, but there are also data structures for SQUARE and TRIANGLE which also have* `SIZE` *components.*

Alas, the programmers who write software this way aren't trying *to be confusing. The problem is that the programming path of least resistance when using data structures, type definitions, and #DEFINE statements is to write code that requires a long trail of bread crumbs to find where its variables actually come from.*

My own choice is in Section 4.2 (page 10); one should try to make all variables character, integers, floating point, or arrays of these and keep those declarations in a small number of global header files.

Another example of Type Two scope confusion is putting a bunch of program parameters into a data structure and passing it, or its memory address, to various functions around the program. As the reader of such code, it is a terrible, twisted maze to figure out whether there is just one set of parameters or if copies are being generated and modified for local use.

A less obvious example of this sort of thing is in the realm of physical dimensions. The variable is called LENGTH, but is it centimeters, inches, cubits, feet, links, meters, rods, chains, furlongs, kilometers, miles, or nautical miles? My own solution is to run every input and output through my dimension definitions.

```
#define KM          (1.0)
#define MILES       (1.609334*KM)
#define METERS      (0.001*KM)
#define FEET        (0.000189394*MILES)
#define YARDS       (3.0*FEET)
#define FURLONGS  (660.0*FEET)
#define NM       (1853.3*METERS)
```

While some of these items may fall out of the scope of what is traditionally called scope in programming, they certainly should be mentioned and I think of them in the same breath as the regular variable scope.

9.10 *Operator Overloading*

Operator overload is an extreme case of making the reader hunt all over the place to figure out what you were trying to do, another kind of Type Two scope confusion. The arithmetic operators should do the obvious things for all kinds of integers, floating point values, and complex values. Anything else strikes me as gratuitous rather than illuminating. Defining the plus sign for set union and the asterisk for set intersection, for example, is far more confusing than defining functions like UNION and INTERSECTION to operate on sets. (My own preference would be to have integer indices to refer to the sets in question rather than use set classes or objects anyway (page 10).)

I admit there is the occasional time, for example, one is working on quaternions and it really is easier to read `A+B*C/2.0` *than* `PLUS(A,TIMES(B,DIVIDE(C,QUAT(2.0))))`*. At least I wish they would let me put two argument functions in the middle like Dartmouth BASIC did with DIV and MOD so I would write* `A PLUS (B TIMES (C OVER QUAT(2.0)))`*. It isn't quite as clear as* `A+B*C/2.0`*, but it would be nice to have once in a blue moon. Not once in my entire career, however, has the option of operator overloading seemed attractive to me.*

I admire FORTRAN 77 for its decision not to overload the plus sign but to use the double slash (//) for string concatination.

9.11 *Function Overloading*

I feel the same way about function overloading as I do about operator overloading. FORTRAN has all the function overloading I ever wanted, MIN, MAX, ABS, SQR, and the mathematical functions all work for all the reasonable arguments including COMPLEX and DOUBLE PRECISION COMPLEX. What more do you need?

The standard argument I have heard for overloading functions is that there is some essential notion, usually rotation, that is common among a collection of different things. We rotate triangles, squares, and pentagons (but not circles) and it makes sense, they tell me, to have a common function notation smart enough to call TRIANGLE::ROTATE, SQUARE::ROTATE, or PENTAGON::ROTATE as needed. But somehow I end up looking at `ROTATE(P)` *with no idea what P is and getting no help from the function name, yet more Type Two scope mayhem. I would be much happier with the longer and more explicit names ROTATE_TRIANGLE, ROTATE_SQUARE, and ROTATE_PENTAGON to make it* locally *clear what is being rotated.*

9.12 *Don't Call Us, We'll Call You*

Another trend to be avoided is having functions call various functions in your program depending on what happens to them. So long as they can only call one or two functions, this is easy to manage, but when the choices become numerous, it mitigates the benefit of even the best source code debugger.

The X11 interface (also called X-Window) has a function called XNextEvent that sets `REPORT.TYPE` *to ButtonPress, ButtonRelease, KeyPress or whatever event it is looking for. The programmer does not lose control of the flow while debugging the program. The WIN32 API (Section 14.6, page 73) is a little worse, the programmer calls GetMessage which has, deep inside one of its structures, a function that GetMessage will call when an event occurs, still not too terrible I have been told that it is popular to have an event detector call one function for a mouse click, another function for a mouse release, and yet another function for a keystroke. Now what happens to the programmer expecting a mouse release when something unexpected happens and the program thinks a key has been pressed?*

9.13 *The Great Quest*

FORTRAN was a major achievement in programming technology. There is a completeness to it, especially in its 1977 version, that makes me sit back as after a tough climb to a beautiful view. Sure, there are one or two things one might add in hindsight, and my list may be different from yours, but the language as a package has a totality about it, an essential wholeness. At a certain level of computing content, FORTRAN 77 can say clearly all that needs to be said with a small vocabulary used precisely and exquisitely.

I remember reading an article about Baccus[3] where he discussed the quest for the next level of programming. I remember he expressed his frustration that there seemed to be no reasonable way to codify in programming language meaning and structure beyond *what FORTRAN could say. Attempts to do so created artificial constructs that did not present the programmer with anything approaching a balanced harmony of organization.*

Maybe it isn't there. Maybe FORTRAN really articulates all the things that the programming language itself should be saying and the rest is up to the programmer. After all this time, I'm confident that the people currently looking for it are not going to find the next level of programming. And just think what that same effort would have produced for all of us if the same effort had gone into making programs better rather than perpetually inventing new ways to write programs.

But is it so terrible to leave beauty and structure up to us? If Leonardo da Vinci and Rembrandt van Rijn could produce the beauty they produced with canvas and a few colors of paint, maybe we can find meaning and truth within the vocabulary of FORTRAN. I have been doing exactly that as my profession for twenty-one years, long enough to be comfortable with it, and I haven't felt the compulsion, or the obsession, to find a "new and improved" way of writing programs. Rather I have been satisfied, even enthusiastic, looking for new and improved insights into the very hard and very deep problems whose solutions will add value to my employers.

9.14 *Software Aesthetics*

What constitutes a good looking *program? When does source code look* pretty*? I have to admit that my own sense of beauty in all things is severely shaped by what works well. I love the Porsche 930 which may be noisy and funny shaped because it goes like the devil and lets the driver be at one with his machine. In the realm of electronic switches, my sympathy was always with the 1A ESS which was analogue and noisy, with all its reed switches clattering, because it switches a whole lot more calls than the new generation of digital switches and it doesn't have all the software bugs that characterize the new technology.*

I have similar attitudes in software. While I understand *the aesthetic sense that draws others to data structures, memory pointers, objects, recursion, and all the other post modern techniques, I see the degradation of function that goes along with them, degradation of run-time efficiency and of understanding and maintenance. While others have been "seduced by the dark side" of programming, I have remained unconverted.*

When I see a neat set of parallel arrays in one or two header files run in a well organized set of functions without recursion, I feel I'm looking at the Rembrandt of software. I keep my FORTRAN cellular simulation from 1984 as an example of beautiful code, and it was the very first to employ the Adam style I have used for the nineteen years since. It's neat, it's clean, it's fast, it's sweet, it's just right. Adding post modern constructs would sully it.

10 COMPILER DIRECTIVES IN C

Some of the most wonderful features in C are the compiler directives #IF, #ELIF, #ELSE, and #ENDIF. I no longer have to feel guilty writing debugging code with an IF statement around it knowing that valuable CPU time is being wasted evaluating the IF condition. I can have a sequence of #DEFINEd variables DEBUGA, DEBUGB, DEBUGC, . . . most of which are set to zero at any given compilation.

I have real problems when they are used as code editors. For example, consider a loop that compiles one way with HALFWAY set to one and another way with HALFWAY set to zero.

```
#if HALFWAY
for (jl = 1; jl <= numl/2; jl++)
#else
for (jl = 1; jl <= numl; jl++)
#endif
{
  kl = numl-jl;
  linetest (jl, kl);
}
```

I find this code nearly impossible to read. No combination of indents or comments will make me able to concentrate on the loop itself. If I were making the distinction during program execution, I would code it this way

```
if (HALFWAY)
{
  for (jl = 1; jl <= numl/2; jl++)
  {
    kl = numl-jl;
    linetest (jl, kl);
  }
}
else
{
  for (jl = 1; jl <= numl; jl++)
  {
    kl = numl-jl;
    linetest (jl, kl);
  }
}
```

and have no trouble understanding what was happening. And my rule for #IF statements is that I always code them and indent them like ordinary IF statements. After all, when I read code do I really care whether a decision is a compile time or run time decision? No, I only care what the code is going to do when I run it. So I bite the bullet and type the loop twice.

```
#if HALFWAY
  for (jl = 1; jl <= numl/2; jl++)
  {
    kl = numl-jl;
    linetest (jl, kl);
  }
#else
  for (jl = 1; jl <= numl; jl++)
  {
    kl = numl-jl;
    linetest (jl, kl);
  }
#endif
```

There is an important but subtle point to be made about C macros as compared to subroutines. Consider the following function ISUM that returns the sum of J and K if ITEST is true.

```
                    /* ***** ISUM ***** */
int isum (itest, j, k)
int       itest, j, k;
{
  if (itest)  return (j+k);
  else        return (0);
}
```

Further, consider the case where I call subroutine ISUM in the middle of my tightest loop. I hate the idea of throwing three INT values on the stack and reading one off the stack every time I call ISUM. So I can use a C macro as follows which uses the ? operator to do the same thing as my old ISUM function.

```
#define ISUM(itest,j,k) (itest)?(j+k):0
```

The interesting bit of confusion comes when my macro is complex enough to require multiple statements and corresponding braces. Suppose I have a subroutine LINE that draws a line from (JX,JY) to (KX,KY) on my computer screen. Further, suppose I have a macro TWO_LINE that draws lines from (JX,JY) to (KX,KY) and from (KX,KY) to (LX,LY). The syntax for a function is different from that of a macro since the braces preclude the use of a semicolon before the ELSE.

```
#define TWO_LINE(jx,jy,kx,ky,lx,ly,kolor)   \
           { line (jx, jy, kx, ky, kolor);  \
             line (kx, ky, lx, ly, kolor); }
  . . .
if (kolor)  line (jx, jy, kx, ky, kolor);
else        line (jx, jy, kx, ky, 1);
  . . .
if (kolor)  TWO_LINE (jx, jy, kx, ky, lx, ky, kolor)
else        TWO_LINE (jx, jy, kx, ky, lx, ky, 1)
```

I mention this case since I have been bitten by it.

11 TOOLS

There is a growing awareness of how useful software can be when writing other software. I've written countless stupid little programs to help me debug other programs. Some of these have evolved into useful tools that I use all the time.

11.1 The RENUMBER Program

FORTRAN uses numbers for labels. While they have almost no mnemonic properties, numbers at least have the advantage that one can easily recognize one as being greater than another. The RENUMBER program runs through a FORTRAN program and changes the labels in each program, subroutine, and function to be 10, 20, 30, *et cetera.* It does not handle arithmetic IF statements or computed GOTO statements.

```
if (x) 40, 50, 50
goto (110, 120, 130, 140, 150) itest
```

In addition to regular FORTRAN label assignments (as seen in my code, anyway), RENUMBER recognizes the construct LABEL 110 in strings and comments and renumbers it along with the code. That allows me to keep debugging audit code in my program.

```
      jl = iloccc (jc, kc, lc)
c                JL should never be zero.
      if (jl.eq.0) then
140     write (*,*) 'Label 140 * Something broke here.'
      end if
```

Since I use tabs exclusively, my RENUMBER program can do strange things if spaces are used instead of tabs for the six character indent in non-commentary non-continuation source code.

11.2 The CTEST Program

I broke down and wrote a program that reads a C program and reports braces with mismatched indentation, single equals sign in a decision expression in an IF, FOR, or WHILE statement, comment starts within comments, mismatched parentheses, brackets, and braces, double equals signs outside IF, FOR, or WHILE statements, *et cetera.* I tailored the program to coding techniques I actually use since the effort of substituting all the #DEFINEd strings and macros was too much to bother with.

11.3 The CNUMB Program

Finally, I wrote another program which goes through a C program and appends to lines ending in a brace the range of lines covered by the loop.

```
#include <stdio.h>
main ()
{                       /* 3-10 */
  int j;
  for (j = 1; j <= 10; j++)
  {                     /* 6-8 */
    printf ("%d %d\n", j, j*j);
  }                             /* 6-8 */
  exit (0);
}                               /* 3-10 */
```

As programming constructs get to be hundreds of lines long, it is a pleasure to see how many lines down the other end is. Even though my editor has a brace matching feature, I still find this gives me information I didn't otherwise have.

11.4 *The HGEN Program*

The DIMENSION and COMMON statements did a great deal in FORTRAN and there are declarations which sort-of do those things in C. But C does allow run time allocation of memory and even a die-hard FORTRAN person like me would be remiss not to take some advantage of it. I establish an array bounds check that can be turned on (`RANGE=1`*) or off (*`RANGE=0`*).*

```
#if RANGE
  #define Range(j,m,n) zrange(j, m, n, __FILE__, __LINE__, #j)
#else
  #define Range(j)
#endif
```

The ZRANGE function gets the actual index value J along with its lower and upper bounds M and N. But a good bounds check reports more than the existance of an infraction, it reports where the infraction occurred and which variable is out of bounds. So we use a C macro called RANGE to throw in the file name, the line number, and the name *of the* *variable which ANSI C lets us do with* `#J` *which creates a quoted string with the text of the argument. So the C expression* `ARRAY(K+7)` *in MYPROG.C in line 93 with* `K+7` *out of range would print*

```
        myprog.c(93):  Value K+7=19583 is greater than 100.
```

which gives the programmer a fighting chance of figuring out what went wrong.

Then we need to define and declare each array with the bounds checking.

```
    double           *fradoc;
    #define fradoc(jc) fradoc[Range(jc,1,numc)]
```

ANSI C is nice enough to let us use the same name for the definition and the variable in this construction, at least in every compiler I have used. And finally we need to allocate the memory itself at the beginning of program execution. I have a function called MAL that allocates the memory with an error message if it fails. Its second argument is a string that is printed in the error message if the memory allocation fails so I can tell how far I got if I run out of memory. (I'll be the first to admit it only helps somebody with the memory allocation source code, but it gives us a fighting chance.)

```
      j = sizeof (double)*maxc;
      fradoc = (double *) mal (j, "fradoc")-1;
```

I have taken a liberty here in using `MAXC` *for the memory allocation and* `NUMC` *for the bounds check. If I allocate memory for one hundred circles and only use thirty-one of them, then an index of thirty-two is out of range so far as I am concerned. It means I have to be excruciatingly careful not to let* `NUMC` *exceed* `MAXC`*, but I catch errors using* `NUMC` *that an old fashioned bounds check using* `MAXC` *would have missed.*

The MAL function has another neat property. In the case of "tall and thin" arrays there will be "zillions" of memory allocation calls. Consider a two dimension array of 2500 legs and eight fleets. Declaration and memory allocation look something like this.

```
    #define fvolf(jl,jf) fvolf[Range(jl,1,numl)][Range(jf,1,numf)]
    double             **fvolf;
         . . .

      j = maxl*sizeof (double *);
      fvolf = (double **) mal (j, "fvolF")-1;

      k = maxf*sizeof (double)
      for (jl = 1; jl <= maxl; jl++)
        fvolf[jl] = (double *) mal (k, "fvolf")-1;
```

This leaves us with 2500 sixty-four byte chunks of memory. A three dimensional array can easily allocate memory tens of thousands of times and some computers are not terribly efficient at allocating memory. So the MAL function allocates anything smaller than 2000 bytes in bigger clumps, 32000 at a time. These parameters can be set at library compile time.

The joy of programming is the formulation of heuristics and algorithms to solve hard *problems. That joy does not include the mindless repetition of memory mangement code knowing that the slightest typographical error will cost me days of painful debugging. So I wrote a Header GENeration utility called HGEN to do it all for me. I create a FOO.G file from which HGEN generates a global header file FOO.H and a FOO_MEM function in FOO_MEM.C to allocate memory. While it is already writing FOO_MEM.C, HGEN also generates FOO_WB to write binary files and FOO_RB to read them back, storing all the array values from one to NUM-whatever. HGEN also generates all the "structure-to-structure" copy functions, FOO_C_COPY for circles, FOO_F_COPY for fleets, FOO_L_COPY for legs, and FOO_LF_COPY for leg-fleet combinations. And just to make life a little easier for me later on, HGEN generates FOO_C_PRINT to print all the "structure components" of circles, FOO_F_PRINT to print all the components of fleets, FOO_L_PRINT to print all the components of legs, and FOO_LF_PRINT to print all the components of legs and fleets.*

The input file FOO.G has the arrays in dot-first lines with the comma after the optional semicolon.

```
. double fradoc ; radius of circle

. double fvolf ; economic value of leg and fleet
```

In case a few extra circles show up and the original `MAXC` *isn't big enough, the MAL function does some internal pointer trickery so it can free all the memory it allocated earlier in a big chain reaction. That means that the programmer should never FREE memory allocated using MAL since the internal reference pointers will not be right. The memory expension code may be awkward, but it gets the job done. And the application programmer using code like this doesn't need to be bothered with managing memory and can get on with doing what I consider to be the* real *job.*

All of this presupposes the programmer is using a great deal of organizational structure. The alternative is to have total freedom of sizes, structures, and pointers and no idea where the next memory byte is coming from. I don't feel terribly constrained to budget sizes in advance and to start counting at one all the time. It may or may not be better for you this way, but I know it is better for me.

There is one other "trick" in HGEN. I wrote an alternative form of its output that converts parallel arrays into data structures. For index arrays (used in QSORT for example) the G file has two dots instead of one. For an example, consider a list of circles with (X,Y) location, radius, and color that we want sorted in our program. This is the input to HGEN in a file called FOO.G.

```
typedef short Circle;
typedef char   Kolor;

.  double   fxoc ; X-coordinate of circle
.  double   fyoc ; Y-coordinate of circle
.  double fradoc ;       radius of circle
.  Kolor    nkoc ; (kute) color of circle
.. Circle   icoc ;   sort array of circles
```

And this is the normal output of HGEN in FOO.H.

```
#define fxoc(jc) fxoc[Range(jc,1,numc)]
EX double       *fxoc; /* X-coordinate of circle */
#define fyoc(jc) fyoc[Range(jc,1,numc)]
EX double       *fyoc; /* Y-coordinate of circle */
#define fradoc(jc) fradoc[Range(jc,1,numc)]
EX double         *fradoc; /*       radius of circle */
#define nkoc(jc) nkoc[Range(jc,1,numc)]
EX Kolor        *nkoc; /* (kute) color of circle */
#define icoc(jc) icoc[Range(jc,1,numc)]
EX Circle       *icoc; /*   sort array of circles */
```

And this *is the output of HGEN in FOO.H if my HGEN program is run with the /STRUCT option.*

```
#define fxoc(jc) foo_c[Range(jc,1,numc)].fx
#define fyoc(jc) foo_c[Range(jc,1,numc)].fy
#define fradoc(jc) foo_c[Range(jc,1,numc)].frad
#define nkoc(jc) foo_c[Range(jc,1,numc)].nk
#define icoc(jc) icoc[Range(jc,1,numc)]
EX Circle       *icoc; /*   sort array of circles */

struct _foo_c
{
  double fx;                                /* X-coordinate of circle */
  double fy;                                /* Y-coordinate of circle */
  double frad;                                    /* radius of circle */
  Kolor nk;                                 /* (kute) color of circle */
} *foo_c;
```

Note that ICOC *is not absorbed into the data structure for circles. This way I can later call a sort subroutine with* ICOC *to sort the circles by whatever criterion is in the ICMP_C function.*

```
        qsort (icoc+1, numc, sizeof (icoc[1]), icmp_c);
```

It may be a hassle to use this tool, but it gives the C programmer all the power and control of FORTRAN with the run-time memory allocation convenience of C. To me this *is a whole lot better than the usual pointers-to-structures nonsense that has become the standard programming approach in C and that evolved into object oriented programming systems.*

11.5 *The IDATE Program*

As much as the world likes long file names, I prefer to keep my file names in the eight-dot-three paradigm of DOS and older versions of VAX/VMS. With a little effort, IBM TSO can be coaxed into cooperating with eight-plus-three file names. I wrote a cute little program called IDATE that generates a three-character string for the date. I use the first two characters for the year and month and the last digit for the day of the month I set 1989 December 1 to be 001, so 1989 December 9 is 009, 1989 December 10 is 00A, and 1989 December 31 is 00X. 1990 January 1 is 011 and, more importantly these days, 2001 October 31 is 46X. I find having a three-character date code is a convenient tool for backup file extensions and other date coding like calling this Halloween's data file FILE-46X.DAT.

12 PROGRAMMING TECHNIQUES

The pedestrian word for programming techniques is "tricks." Here are a few ideas I've used to make life easier in some situations and bearable in others.

12.1 Array Bound Checking in C

One of the features I rely on in most FORTRAN compilers is array bound checking. When compiled with this feature, the following code generates an abnormal termination when J is seven.

```
      dimension a(6)
      do 10 j = 1,7
        a(j) = 3*j
 10   continue
```

I like this feature so much that I have large complex applications that run for hours with this feature still enabled. If I still have code that generates array subscripts out of bounds, I want to get the error message rather than to spend weeks tracking down the strange behavior.

The way I get around this in C when I need array bounds checked is to put the following function into my program.

```
                /* ***** RANGE ***** */
int range (m, n)
int       m, n;
{
  static j_error_count = 0;
  if (m < 1)
  {
    printf ("Range error:  %d is less than one.\n", m);
    j_error_count++;
    if (j_error_count > 5)
    {
      printf ("Too many range errors.\n");
      exit (2);
    }
    return (0);
  }
  if (m > n)
  {
    printf ("Range error:  %d is greater than %d.\n", m, n);
    j_error_count++;
    if (j_error_count > 5)
    {
      printf ("Too many range errors.\n");
      exit (2);
    }
    return (n-1);
  }
  return (m-1);
}
```

The RANGE macro/function either does nothing (for performance) or it tells me when my argument is out of range and quits after five failures. I put the following at the beginning of each source file.

```
#if RANGE == 0
  #define r(x,y,z) (x)
#else
  #define Range(n,nmin,nmax) \
         zrange(n,nmin,nmax,__FILE__,__LINE__,#n)
#endif
```

Normally, we set RANGE to zero (or just leave it out) and the Range macro simply uses its first argument. But we can compile the source file with RANGE set to one and the Range macro calls ZRANGE which does range testing before decrementing its first argument.

The "technique" is to declare arrays this way.

```
#define ncol(jl) ncol[Range(jl,1,MAXL)]
int              ncol[MAXL+1];    /* first circle of line */

#define nccol(jl) nccol[Range(jl,1,MAXL)]
int               nccol[MAXL+1]; /* second circle of line */

#define ntocl(jc,jl) ntocl[Range(jc,MAXC)][r(jl,MAXL)]
int                  ntocl[MAXC+1][MAXL+1];
                    /* one if circle is touched by line */
```

This range checking of array indices is built into the HGEN program described in Section 11.4 (page 62) which uses memory-allocated pointers instead of explicitly declared arrays. When RANGE is zero, the argument is used without modification. When RANGE is one, each subscript is checked to see if it is in bounds. Another advantage of this for a FORTRAN hacker is that arrays are now denoted by parentheses with commas between subscripts like I am used to. I have had no trouble using the same name for the array macro and the actual C variable name.

12.2 While Loops in FORTRAN

Most FORTRAN compilers have a DO WHILE statement

```
      itest = ITRUE
      do 180 while (itest.eq.ITRUE)
         . . .
180   continue
```

and some even have an END DO statement.

```
      itest = ITRUE
      do while (itest.eq.ITRUE)
         . . .
      end do
```

Enough compilers do not have a DO WHILE statement that I avoid it. My usual syntax for WHILE loops uses an IF-THEN construct.

```
      itest = ITRUE
180   if (itest.eq.ITRUE) then
         . . .
        go to 180
      end if
```

12.3 Calling by Value in FORTRAN

FORTRAN passes addresses of arguments to subroutines and functions. If the argument is an expression, then the value is computed, stored, and passed by address. In this example, the value of I is changed to seven while J remains unchanged.

```
      program exampl
      i = 3
      j = 4
      call change (i)
      call change (j+1)
      write (*,*) i, j
      end
c
      subroutine change (k)
      k = 7
      return
      end
```

I will sometimes put an extra pair of parentheses around an argument to make the compiler treat it as an expression. According to the standards, neither I nor J should change.

```
      program exampl
      i = 3
      j = 4
      call change ((i))
      call change (j+1)
      write (*,*) i, j
      end
c
      subroutine change (k)
      k = 7
      return
      end
```

However, I have found new-school compilers that ignore extra parentheses under the assumption that they are merely the product of a deranged programmer. So if you take advantage of this feature of FORTRAN, check your compiler out carefully.

13 SOFTWARE REUSE

One of the hallmarks of the new age is that we treat working software as something precious. When I need a piece of code to do something, I usually just write it. Sometimes there is a program that already does what I want; but usually it just isn't close enough to bother with.

Let me give an example. I recently had a set of keywords on separate lines in a file. I wanted a FORTRAN subroutine that would discriminate among them in a tree type search: Separate cases by first letter, then by second letter, and so on. In a half hour, I had a program written that read the list of 190 keywords from INPUT.DAT and wrote my 1300 lines of FORTRAN code into OUTPUT.DAT. I felt I had accomplished something by saving myself the labor of typing 1300 lines of code.

The new view is that I should have used a tool called LEX (which I have never used before) that some say could have produced C-code to do the job that I could easily have converted into FORTRAN somehow. After all, if software already exists to solve the problem, why write new code?

My answer is that wheels may be interchangeable, but I'm not going to whittle a wheel into an airplane wing. Rather, I'll start with a clean slate and design the wing. Similarly, I'll reuse code when applicable and I'll design my code so it can be reused. But the notion that software is too precious to be thrown away, that writing code for tools is the last resort of a desperate person, is not consistent with my views.

14 IDIOSYNCHRACIES

In a perfect world, all compilers would create perfect executable from ANSI standard code and perhaps compile some useful superset of the standard features. Documentation and reality would always match. And none of the energy of the programmer would ever be diluted debugging compilers or operating systems.

There are occasions where these criteria of perfection are violated. Some of these violations are worse than others but all of them can require knowledge of compiler and operating system internals that should not be part of my job.

The list here is not exhaustive, but it is representative. Familiarity with these kinds of bugs (or "features") should make you more aware of where to look when "it should have worked perfectly."

14.1 UNIX

My favorite bug in the AT&T UNIX FORTRAN compiler, F77, is that it has hidden reserved words. Standard FORTRAN has no reserved words at all. If you want to name variables `DO 100 I`, `REAL`, `COMMON`, and `FORMAT`, then you go right ahead and do it. But a piece of code failed during a file OPEN statement when the programmer had an innocent looking subroutine called SCAN. The code ran fine when he changed the name from SCAN to ESCAN.

The conclusion we drew is that SCAN is an undocumented internal routine called during the opening of a file. Of course, if such internal routines are in the library, then they should have names with dollar signs or underscores so no standard FORTRAN subroutine name can conflict with it.

The lesson to be learned is less clear. Certainly, we can't learn to avoid a set of names without knowing what they are. The lesson I learned is that, in the UNIX environment, strange behavior may be eliminated by changing subroutine names.

The C compiler in AT&T UNIX has no function prototyping. We can still declare a function's return type.

```
void  organize ();
int   icolll ();
float fttoc ();
```

But we get syntax errors when we declare function arguments.

```
void  organize (int *, int *);
int   icolll (int, int, int);
float fttoc (int);
```

14.2 C

There are several traps waiting for C neophytes. I make no claims of completeness, but here are four examples.

The most obvious error that is inherently hard to track down is forgetting ampersands in SCANF statements. Suppose we are trying to input the values of integer variables J, K, and L. The statement

```
scanf ("%d %d %d", j, k, l);
```

looks reasonable and compiles without complaint. The most obvious effect is that J, K, and L never get changed by it and strange side effects occur depending on the values of J, K, and L. The problem is that SCANF writes to the *address* specified by its argument and we are passing it the *values* of J, K, and L. The correct statement

```
scanf ("%d %d %d", &j, &k, &l);
```

sends SCANF the address we want written.

Sometimes spaces are required in C expressions. A fellow programmer wanted a quotient of two pointer values. He had code that looked like this.

```
float *a, *b, *c, *d;
*c = *a/*b;  /* Divide A by B to get C. */
*d = *a+*b;  /* Add A and B to get D. */
```

The compiler saw the slash-star combination in the middle line as a comment indication with results you can imagine.

There are numerous clever tricks you can play with the notation for increment (++) and decrement (–). The four statements

```
i = j---k;
i = j- --k;
i = j-- -k;
i = j- - -k;
```

might all do different things.

So when does one stop being a neophyte? When can one expect to graduate from being a Lowly C Novice to being an Exalted C Programmer with the respect that rank deserves? My experience and the experience of others suggests that somewhere around 50 thousand lines of debugged code is about how much experience it takes to get past these annoying problems. If you do all your work on a single machine and compiler and seldom port code, then your comfort level may rise sooner as long as you never change machines or compilers. On the other hand, lines spent trying to use every feature of the language no matter what the application requires should not be counted toward the 50 thousand lines.

14.3 Alliant

As many of my friends already know, I've done some programming on an Alliant "super-mini-computer." Let me offer a word of warning to others.

Their FORTRAN compiler has hidden reserved words in it, passes parenthesized constants by address, and has more subtle behavior when code is optimized for its vector and concurrent modes.

Their C compiler gave me a hard time running a program that computes the distance between two latitudes and longitudes until I found out that the COS function was returning strange nontrigonometric values.

It seems today that using a machine with a particular area of excellence means taking extra effort to make sure the more mundane areas have been covered.

14.4 ANSI-C

I avoid passing FLOATs; instead I pass pointers to FLOATs. It sounds stupid, but there is a conflict between the original C language which promoted all FLOATs to DOUBLEs when passing them to subroutines and the ANSI C which passes FLOATs as FLOATs. Compiler writers have to sit on the fence and promote FLOATs when subroutines are declared the old way and leave them alone when subroutines are declared the new way. The eventual result of passing FLOATs is total confusion and countless strange errors that take weeks to debug. So if I really want to pass a FLOAT rather than a DOUBLE to the PCOMB function, I do it the following way and I don't get into compatibility trouble.

```
#include <stdio.h>
#include headers.h

float pcomb (float *, float *, float *);

                /* ***** MAIN ***** */
main ()
{
  float x, y, z, xyz;
   . . .
  xyz = pcomb (&x, &y, &z);
   . . .
}

                /* ***** PCOMB ***** */
float pcomb (pa,  pb,  pc)
float       *pa, *pb, *pc;
{
  float a, b, c, abc;
  a = *pa;  b = *pb;  c = *pc;
  abc = (a*b*c)+(a*b)+(a*c)+(b*c)+a+b+c;
  return (abc);
}
```

14.5 LAHEY

One feature of the Lahey Computing Systems FORTRAN compiler that runs in Intel 80386 "protected mode" is that it writes in the executable file compiled code and data space for DIMENSIONed arrays. The good news is that COMMON arrays do not required extra disk space. As a result, I tend to put large arrays in COMMON even when they are not referred to anywhere else in my code. This often rewards me when I take one large main program and divide it into subroutines all using the same data, since I already have a header file with COMMON statements.

14.6 *Windows*

The Windows 32 bit Applications Programmer Interface, the WIN32 API, has more than a few idiosynchacies for the C programmer. The big one is that PRINTF doesn't work. Never mind how many windows are open or whatever interface they're using, PRINTF should be able to print somewhere. I had to write a whole text output system with end-of-line wraps and screen scrolling. It's a mess.

Another strange Windows feature is that the main function is called WINMAIN instead of MAIN. While it isn't a big deal to have two source files, one for Windows and one for *something else, it would seem a reasonable concession to standardization and consistency to use the same main program name as everybody else.*

14.7 *IBM TSO*

I spent some time using the SAS C compiler for the IBM TSO mainframe where everything is different from DOS, UNIX, VAX/VMS, and anything else I can think of. The people who wrote that compiler did a terrific job of adapting a stream-of-bytes terminal interface language to the mainframe.

The fundamental 3270 terminal interface is unique. The user may not type when the terminal is printing and the computer can print only system messages when the user has control. When the terminal fills the screen, the user presses a key to clear the screen and resume typing from the top, so there is no scrolling capability. I had to write my own subroutines to replace some of the "lost" functionality of the 3270.

File names are peculiar with regular datasets, partitioned datasets, and Data Dictionary (DD) names. A regular file name USERID.TOP.MIDDLE.BOTTOM has the six letter user identification followed by alphanumeric strings up to eight characters each separated by dots. There is no obvious connection between files with the same strings, so USERID.FRUIT.APPLE and USERID.FRUIT.BANANA have nothing in common by virtue of the common word FRUIT.

Partitioned datasets have files inside them, denoted by parentheses. My source code was in USERID.SOURCE.C and my headers were in USERID.SOURCE.H, so #INCLUDE <TOP.H> would send the compiler looking for USERID.SOURCE.H(TOP). For the VAX, DOS, UNIX veteran it takes some getting used to.

Finally, there are DD names, the single words assigned to files, either regular datasets or members of partitioned datasets. These are used frequently in CLIST or JCL scripts. The SAS C compiler was kind enough to work well on all of these.

Remember Job Control Language (JCL)? The command line seen in `ARGC` *and* `ARGV` *in the MAIN function is described by* `PARMS=` *on the JCL declaration. How about ISPF, the IBM mainframe answer to the MacIntosh? I was able to write interactive screen sessions by calling ISPF in seriously non-standard ways.*

But the big difference is the EBCDIC alphabet instead of ASCII. The EBCDIC alphabet is not contiguous, there are non-alphabetic characters between 'A' and 'Z' which is not *the case in ASCII. Also, the digits are greater than letters, so '0' > 'A' which is the opposite of ASCII. The best defense against these difficulties is to stick with the CTYPE functions ISALPHA, ISDIGIT, ISALNUM, ISSPACE, ISPRINT, and ISGRAPH and do not compare characters to 'A' and 'Z' to see if they are letters. Both ASCII and EBCDIC have the digits zero through nine increasing incrementally, so '5' is '3'+2.*

14.8 *IBM C Compiler*

Of all the companies I would not *expect to screw up the standards game, I would have put IBM on top of the list. But they, too, have thumbed their noses at standards.*

```
#include <stdio.h>
int main ()
{
  char q[80];
  printf ("What is your name? "), gets (q);
  printf ("     Your name is %s.\n", q);
  return (0);
}
```

The program is supposed to print "What is your name?" and then it is supposed to sit on the same line *and wait for the input. The SAS C compiler I used to use on the mainframe did it right, even on the horrible 3270 terminal interface. But IBM's compiler goes to the next line to wait for input.*

Is this so terrible? Actually, yes it is. I wrote an entire display interface system for the 3270 to make it print and scroll and interact with a user and now none of it works anymore because it prints an extra line which means the user has to clear the screen before he can type his response. I suppose somebody *can go in there and fix it, but why couldn't IBM just follow the rules of the C programming game?*

15 *THE NEXT GENERATION*

I believe we are at a critical fork in the road vis á vis teaching people how to program. We have some tough decisions to make in the next five to ten years and there is every reason to blame today's teachers for today's educational failures.

15.1 *The Great Decline*

In 1989 they had a small exhibit on the New York World's Fair site in Queens called "Remembering the Future" where they celebrated the futuristic visions of 1939 and 1964. The 1939 future mostly came true but, somehow, the 1964 future fell short of expectations and there are myriad reasons for this including political changes and social evolution. I have felt these changes in several fora but nowhere has the shortfall been so dramatic as in the forum of the digital computer.

As I have done, one may speculate at length as to the increases in computer hardware, our investment in computers, and the gain we should realize as yesterday's software adventure becomes today's routine. And one may similarly speculate on the computer's value-added over the last two decades. I suggest we should consider the social contributions of electronic mail and the World Wide Web as the product of universality rather than programming efficacy. And the suite of spreadsheets, word processors, and databases may be ubiquitous, but do they really add value? Their lack of reliability alone limits their contribution compared to pencil and paper.

So we have invested a lot more with better machines in a task that should be easier and have come up with less. I'm willing to concede most of that to bad corporate manage-

ment, lousy education, social irresponsibility, and mismanaged expectations. (Is there any institution I haven't criticized in that sentence?) But there is still an enormous shortfall in the programming arena.

I believe much of the change is an attitude change. Two of today's most renowned computer scientists told us, "Most users of a tool are willing to meet you halfway; if you do ninety percent of the job, they will be ecstatic."[7] I am horrified by this mandate and more horrified by the results it has spawned. I am embarrassed when my user finds a case I didn't think of, which does *happen from time to time in spite of my best efforts. It should be a point of pride in* every *programmer that his tools work first time and every time for a long time.*

On the occasion that there is a very rare case that is very hard to deal with, the tool should detect it and announce it. And if that is demonstrably too hard, with the burden of proof on the programmer rather than the user, then the rare case should be documented and explained to the user.

The other fora that come immediatly to my mind are hifi and running. In hifi, a genuine and scientific inquiry into our preception of music has degenerated into a sea of expensive and highly colored musical instruments masquerading as genuine high fidelity, The running movement, which lured a generation away from its television sets into the glory of the Great Outdoors has degenerated into a generation of health club "potatoes" on treadmills watching the same television shows we were so happy to run away from. The results in running were predictable and are measureable: In 1979 there were 29 high school runners who ran two miles under nine minutes and in 1999 there were none equivalently fast for 3200 meters. So it isn't just *computers and software that are suffering.*

I bring this Great Decline up for two reasons: urgency to remedy a growing problem and recognition that today's software methods have failed.

In a sea of computers playing games and performing office functions, there is a small computing community supporting decisions, mostly business decisions with financial consequences. I have observed a decline in the quality of these decisions over the past few decades. As our economy employs more people for longer hours to produce less, I have concluded that my own observations are representative of a larger whole. While it may be self-serving, I believe that the largest value added of the digital computer is in its ability to support business decisions by reporting, simulating, analyzing, and optimizing complex systems in a vast sea of information. In short, computers add value by helping people get better answers to hard problems or to a very large number of easy ones. And it is in the decisions support function that we have seen a dramatic decline in the contribution of the computer.

We have more people writing more software than before, but closer inspection usually shows these projects taking shortcuts that lose sight of the important aspects of the problem being solved. I concede there are many problems amenable to the simplifying assumptions of simulation and optimization packages, but there are just as many that demand the focus and precision of a programmer who really understands the internal processes of the decision being studied.

Should an airport have one or two ticket counters? Does a railroad's freight traffic *justify building more track? Does an apparently clever radio communication design offer its theoretical capacity improvements in practice? How can we use image processing software to support or automate millions of map based decisions? Are early predictions of winds-aloft good enough forecasts to justify changing weight limitations for airline ticket sales? What are the technology tradeoffs for radio telephony and how are these different for mobile* vs. *stationary subscribers? Does a sales promotion generate new business or merely get the same customers to buy the same product sooner and for less money? In the longer term, do sales promotions train customers to wait for a sales promotion before making a purchase? What is the effect on a workforce of apparently random "management incentive programs" (MIPs) for early retirement?*

All of these questions offer tremendous economic benefit from greater understanding and require in depth study for meaningful answers. Doing half the job or doing the whole job half as well provides far less than half the benefit of a full study, if it even produces any benefit at all. These and a thousand other tough questions present an urgent need for problem-specific software. That is what I have done for my living for twenty-one years.

I believe the true "value added" of the digital computer is the ability to answer questions like these. More than a change of medium, these are questions that could not be addressed without modern machinery. And, sadly enough, we're not answering them as well as we did twenty years ago.

15.2 *What We're Losing Today*

There is a principle that says that if you really can do it then you can teach it. Let's consider the third FORTRAN program most of us learned to write, a stupid sorting program that takes the size of a list (up to ten) and a line with that many numbers on it and prints the same list in numerical order.

```
      program mysort
      dimension array(10)
c
      write (*,*) 'How many items do I sort?'
      read  (*,*) n
      if (n.gt.10) then
        write (*,*) 'Ten is my maximum.'
        stop
      end if
c
      write (*,*) 'Give me the list:'
      read  (*,*) (array(j), j = 1,n)
c
      do 20 j = 1,n-1
        do 10 k = j+1,n
          if (array(j).gt.array(k)) then
            temp     = array(k)
            array(k) = array(j)
            array(j) = temp
           end if
   10   continue
   20 continue
c
      write (*,*) 'The sorted list:'
      write (*,*) (array (j), j = 1,n)
      stop
      end
```

Outside of the actual algorithm, there is very little you have to explain to a non-programmer for him to understand what this code does. There is the DIMENSION statement so that ARRAY's subscripts go from one to ten and there is the "(,*)" construct after the WRITE and READ statements. That's about it.*

Now consider the same program in old fashioned C.

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  int j, k, n;
  float array[11], temp;

  printf ("How many items do I sort?  ");
  scanf ("%d", &n);
  if (n > 10)
  {
    printf ("Ten is my maximum.\n");
    exit (1);
  }

  printf ("Give me the list:  ");
  for (j = 1; j <= n; j++)
    scanf ("%f", &array[j]);

  for (j = 1; j < n; j++)
  {
    for (k = j+1; k <= n; k++)
    {
      if (array[j] > array[k])
      {
        temp     = array[k];
        array[k] = array[j];
        array[j] = temp;
      }
    }
  }

  printf ("The sorted list:\n");
  for (j = 1; j <= n; j++)
    printf (" %f", array[j]);
  printf ("\n");
  return (0);
}
```

This rendition is pretty minimalist in its use of C obfuscation. While I could have used J=J+1 instead of J++, I feel that the C increment and decrement operators are an intuitive and natural part of programming vocabulary.

But what about the first two lines, the #INCLUDE statements? Already we have *two lines to explain that have no contribution to the student's understanding of what the program is actually doing. Then how do we explain the function MAIN to the student who has not seen a function yet? And what about the dimension of eleven? This will please Spinal Tap fans[11] but will bewilder budding programmers. The alternative, trying to explain that C programmers count to five by saying "zero, one, two, three, four," is considerably more confusing than just explaining that C needs an extra element, at least sometimes.*

I have made the simplifying assumption that SCANF is kind enough to remember its place on the line from one call to the next. The program becomes enormously more complicated if we require the program to use FGETS and parse a string manually in case the compiler isn't so generous in repeated SCANF calls.

Okay, what is my point? That programming has bumps and pitfalls for the novice? That C is worse than FORTRAN in this regard? Well, yes and yes, but not really. The things that make writing this C program a pain in the neck, and not just for the novice I might add, are things that have nothing to do with the sort algorithm itself. And, more to the point, the sort of bright, analytical, logical-flow person I want *to attract to the realm of programming is going to be put off by all this administrative overhead far sooner than process weenies who will generate tens of thousands of lines of mindless code and never bother to ask* why*.*

As the next step in this exercise, I invite the reader to construct a simple sort program for the novice using the pointer-structure programming style that has become popular and then to try to envision such a program written in C++ that is readable by a novice.

This is not an isolated example. As the algorithm issues become more interesting, the current crop of programming tools require more and more attention to less and less important details compared to programming pedagogy a few decades ago. Look at models using arrays, or tables, of two or more dimensions. And once we start teaching students to use recursion in their software solutions, we send them down a long road of muddled confusion where those who feel comfortable are not *usually the ones who understand the issues.*

Whatever the reader envisions in this area is unlikely to be as bad as what has actually happened. The teaching of this sort of programming to novices has produced a post-modern generation that has no sensitivity to what I feel are critical programming issues. The modern vision of Correct, Clear, Consistent, Cheap software has been tossed aside. I hope you don't need me to show you horrible examples of poor programming education; there is a consistent pattern of mismanaging memory allocation and losing track of algorithm flow much more with recently educated programmers than there used to be with young programmers.

15.3 *What Should Be Done*

First of all, I think it is high time to bring BASIC and FORTRAN back into the teaching arena. I don't mean Visual Basic® or FORTRAN 90, but good old fashioned Dartmouth Basic[2] and FORTRAN 77. These languages expose the student to the fun of program-

ming without enmiring him in a quagmire of administrative overhead, memory management, and confusing constructs as in C, C++, Java, or Pascal. The student who finds himeself loving these languages will grow into the programmer I want working by my side in ten years. FORTRAN, in particular, has a small vocabulary that communicates much by being used precisely and exquisitely. When these young programmers learn C they will retain an aesthetic sense aligned with getting useful work from the digital computer.

Second of all, let's put the program back into programming education. The student shouldn't touch a data structure, a memory pointer, or even a function call until he is comfortable with sorts, searches, hashes, heaps, indices, linked-lists, matrix arithmetic, and the other stuff that we used to teach. "Never search what you can sort, never sort what you can heap, never heap what you can hash, and never hash what you can index." I like writing programs and I have enjoyed the application of the programming art to hard problems since I was a high school sophomore.

Third, we should create a generation of algorithm warriors. The zeal of victory over a hard problem can be experienced with the same enthusiasm as an athletic triumph. When a really terrible problem is terrible in many dimensions and all of these succumb to a single clean solution, I savor the joy of crushing domination of my domain and doing a victory dance on top of my cubicle desk. The joy of battle we consider healthy in sports, American football in particular, can be just as healthy in conquest over nature's programming challenges.

16 WHAT I HAVE NOT SAID

There are many strong opinions of mine not in here. The same strength of opinion I have on programming style I have on editors, operating systems, and user interfaces. And where you might have some sympathy if my views here disagree with yours, I expect you'll have a great deal less sympathy for my similarly old fashioned views in other areas. Those who work with me know where I stand on these and on more human issues.

More specifically, this document recommends no particular environment, language, or product. While I think these issues are vitally important, they are the subject of discussions elsewhere. I will advise anybody selecting an environment to consider the specific job that needs to be done and the tools and support offered along with a particular compiler or operating system.

While I may be cynical and pessimistic about the general state of code written today, there is a wonderful library of software support tools from GREP (global regular expression print) to MAKE (that selects only changed files for compilation). These tools are becoming more plentiful, becoming available on more and more platforms, and are making life easier. I'm also comfortable writing my own tools when I need them.

The one truly wonderful tool that persistently comes to my mind as I write this is a source code debugger. I've written code without one and so can you, but why bother? Such good debugging tools are available today in almost any programming language that any programmer should be using them. I've used the six-color screen debuggers and the command line driven debuggers and been pleased with both.

17 CONCLUSIONS

The only conclusion to draw from all this is that computer programming is not terribly different from photography or race car driving: There is a lot of science in it but there is still room for art.

My main mission in writing this is to communicate an attitude toward programming that reinforces the importance of getting the little things right. Variable names, spaces, and comments are little things that supposedly have no influence over the ultimate compiled executable file. But these and other minutae make a staggering difference in the quality of a programmer's life and in the long term impact of a programmer's work.

References

[1] `http://www.gnu.org`

[2] `http://www.digitalcentury.com/encyclo/update/BASIC.html`

[3] John Baccus wrote a lovely article, which I cannot find, describing his satisfactions with the decisions made in the first generation of programming and his comfort that the sequential-programming job was done with FORTRAN. He felt it was time to move on and he described his quest for a bigger and non-sequential approach to problem solving.

[4] `http://www.tuxedo.org/ esr/intercal`

[5] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables.* National Bureau of Standards, 1964.

[6] Forman S. Acton. *Numerical Methods That Work.* Harper and Row, 1970.

[7] Kernighan and Plauger, *Software Tools*, Addison-Wesley, 1976, page 136. *"Most users of a tool are willing to meet you halfway; if you do ninety percent of the job, they will be ecstatic." From an otherwise excellent book I have taken this horrifying quotation. That its authors created the early versions UNIX and C suggests that they took this quotation to heart. In spite of this one bad example, you should read the book and you should take* most *of its advice.*

[8] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes, The Art of Scientific Computing.* Cambridge University Press, 1986.

[9] S. J. Simon. *Why You Lose at Bridge.* Cornerstone Library Publications, 1946.

WARNING - THIS DOCUMENT CONTAINS PERSONAL OPINIONS!
Do not be alarmed if you disagree with something in it.

[10] R. E. Stone. Personal communication, 1980.

[11] The movie “This is Spinal Tap” is a parody of the rock documentaries that were popular twenty years ago and it has a priceless scene where a band member was extolling the virtues of his guitar amplifier because the volume knob went all the way to *eleven*. Unlike all the other amplifiers which couldn’t get any louder when they were turned up to ten, this one could go one more.