

list for entries less than or equal to the square root of the largest number in the list (we discussed the reasons for this earlier). Since the largest number in your list is 999, its square root is approximately 31.607, so the largest entry in the top of the array for which you have to cross out multiples is 31. You can either test each entry you take from the top of the list, and stop the procedure when you find one greater than 31; or you can determine, from the way you filled the list, at what position 31 was stored, and then just run your outer loop to loop at that many positions. Be careful not to try to take the remainder of entries (such as the old value of 9) that have already been set to zero. Why?

When you have finished this procedure, your original list will contain only zeros and primes. Use this information now to fill a *newarray*, one which has 2 (a prime) as its first entry, and then includes all of the nonzero entries from your array. Store these primes in the new array with no “gaps” in between; that is, fill successive locations in the new array with the primes you find. There should be 167 odd primes, plus the value of 2, making your new array 168 entries long.

Print out the primes in a compact form. That is, do not print them out one per line, but rather across the page, say, 10 or 15 or 20 primes per line.

**15.** Prime factors. Use the array of primes created in # 14 to solve the following problem (that is, tack this problem on to the end of that program for now; soon you will learn how to output the results of one program to disk so that it can be used later by another program). Carl Friedrich Gauss proved in 1801 that every positive integer greater than 1 can be expressed as a unique product of primes. Since you now have an array of all of the primes from 2 to 1000, you can use this to determine the prime factors for numbers in that range. Use your array to find and print out all of the prime factors for 5 different integers in the range 10 to 1000. You may either read these integers in, or set them up in a DATA statement. If a number has a prime factor that occurs more than once, be sure to print out all its occurrences. For example, 28 has the prime factors 2, 2, and 7. Make your output meaningful, as usual.

**16.** Goldbach’s Conjecture. In 1742, Christian Goldbach (1690–1764) conjectured that every even number greater than 2 is the sum of two primes (for example,  $20 = 17 + 3$ ,  $88 = 5 + 83$ ,  $7000 = 3 + 6997$ ). This conjecture has not been proved or disproved. Using the array of primes from exercise 14, test Goldbach’s hypothesis for all the even integers from 6 to 200. For each even number in the range, find the *first* pair of primes from your list that sum to it, and print them out. You may have to use a prime with itself (e.g.,  $6 = 3 + 3$ ), or with another prime larger than itself. There may be several pairs of primes that sum to a particular even number, but you only need to see one. Print out a table of the even numbers and the primes that sum to them. Include in your program (even though it should not get executed) logic to test to find if no pair of primes can be found to sum to a particular even number; if no pair is found, arrange to print out the message “Goldbach refuted.”

- 17. Modify the perfect number problem (exercise 7 at the end of Chapter 4) so that you *save* all of the factors of a candidate perfect number in an array; if the number turns out to be perfect, print it out *along with* all of its factors.
- ◆ 18. Set up a “reference array” containing the number of days in each month. Use a DATA statement, such as:

```
INTEGER MONTH(12)
DATA MONTH/31, 28, 31, 30, . . . /
```

Now make use of this array to calculate how many days old anyone is. That is, read in two dates (mm/dd/yy), a birth date and a current date, and calculate how many days have elapsed between the two dates. Be sure to take leap years into account. To simplify matters somewhat, you may assume that the birth year and the current year are not the same. Print out the input dates, properly labelled (“echo” your input), and the number of days old this person is.

To make your program more sophisticated, allow for the possibility that the birth year and the current year are the same and have your code handle this special case.

- 19. Use the reference array in the previous problem to print out a nicely formatted calendar for any specified year. The input to your program will be the year and the day of the week on which the first of January falls (if you can find a good algorithm to compute day of the week of the first of January, you can simplify the input to just the year). Fill a *character* array with the dates of the month and blanks, and print out, with a name label, for each month of the year. Once you have filled the array for a particular month, your program can determine the day of the week on which the next month begins. Make your output attractive, and label the columns for days of the week. Be sure to take leap years into account.

You may find arrays like these useful:

```
CHARACTER*10 NAME(12), DATE*2(31)
DATA NAME/'JANUARY', 'FEBRUARY', . . . /
DATA DATE/ ' 1', ' 2', ' 3', . . . , '31' /
```

*Note:* Could you find a better way of changing two-digit numbers into their equivalent character strings, perhaps by making use of the CHAR and ICHAR functions?

- 20. Expand problem 19 so that it can print out three or four months across the printer page for a more compact calendar. The simplest way to do this is to use a three-dimensional array, with 12 “pages,” one for each month, fill the entire array, and then set up loops that print three or four of the months across the page at one time.

```
CHARACTER*2 CAL(12, 6, 7)
```

**21.** Write a program which will accept as input 500 items of data that lie in the range from 101 to 1000. Count how many of the values input lie in the ranges 101–50, 151–200, . . . , 951–100. Print out the counts, and display them as a *histogram*, that is, print out across your display as many '\*'s as represent the particular count you are displaying. You may find that an array of '\*'s may be helpful in doing this:

```
CHARACTER STARS(80)
DATA STARS/ 80*'**'/
```

- **22.** Using the technique in the sample program that improved the visual display of experimental data in the chapter, write a program which will print out the following pattern:

```
$  
$$$  
$$$$$  
$$$$$$  
$$$$$$$  
$$$$$$$$  
$$$$$$$$$  
$$$$$$$$$  
$  
$
```

A money tree!

Now expand your program so that it will print the pattern:

```
+  
+++  
+++++  
++++++  
+++++++  
++++++  
++++++  
++++++  
++++++  
+++++  
+++  
+
```

Allow the length of the longest segment to vary up to a maximum of 55.

- 23.** A Shell sort begins with a “gap” of  $N/2$  for a list of length  $N$ , and compares all values that distance apart, interchanging them if they are out of order; it then divides the gap length in half and repeats (until gap = 0). Implement this sort.
- 24.** Another way to create *combinations* is by filling a two-dimensional array, since the values in such an array can be defined as follows (for  $n$  and  $m$  beginning at 0):

$$\begin{aligned} C_{n,0} &= C_{n,n} = 1 & (n = 0, \dots) \\ C_{m,n} &= 0 & \text{for } n > m \\ C_{m,n} &= C_{m-1,n-1} + C_{m-1,n} & \text{for } n < m \end{aligned}$$

Set up a two-dimensional array COMB subscripted for 0 to 20 rows and columns, fill it according to the rules given, and print it out 20 values per line. This will create a table of combinations, or “Pascal’s triangle” (if the zeros above the diagonal are ignored). It also gives the *binomial coefficients* for the expansion of  $(a + b)^n$ , since the  $n$ th line of the table will give the multipliers (coefficients) for the successive terms in the expanded version of the formula. For example,

$$(a + b)^4 = a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4$$

and you will find that the *fourth* line of the combinations table begins with the values: 1, 4, 6, 4, 1 (followed by zeros).

**25.** We have worked with two-dimensional arrays in several examples. Let us now look at an example in which one- and two-dimensional arrays are combined. The Utilitarians (Jeremy Bentham and John Stuart Mill) proposed a calculus in which the desirability of an action could be calculated by determining the relative amount of pleasure and pain it caused. An action with an overall higher pleasure rating was then more desirable. However, Mill improved on this scheme by suggesting that some pleasures are *worth more* than others (that is, they should be assigned higher weightings). For example, he is supposed to have said, “Better a Socrates dissatisfied than a pig satisfied.” Thus, let us suppose that our scientists have found a means of monitoring the pleasure and pain centers in our brains, and so can give a pleasure (+) or pain (-) reading for each instance in question. Another group has determined a set of weights for each pleasure or pain involved in our experiment, reflecting their relative merit.

In this exercise, assume you have a table filled with  $M$  (say 20) readings on pleasure/pain (one in each column) for each different action you might perform, and  $N$  actions (rows); let  $M$  and  $N$  be set in a PARAMETER statement at the beginning of the program, to give it greater flexibility. Also assume you

have a set of M *weights*, reflecting the relative worth of each of the M pleasures or pains (the weights should be decimal values, such as 0.01, 0.2, and so on, and their sum should not exceed 1.00). Write a program which will read in the weights (or you can set them using a DATA statement), then read in the two-dimensional pleasure/pain array, and finally apply the weight vector to each row of the pleasure/ pain array, to determine and print out the final "value" (sum of the M weighted pleasures and pains) for all N possible actions you are contemplating. The decision should then be clear. You need only "search" the resulting array of weighted results for the largest, and you will know which action to take.

*Note:* This procedure also has applications to applying weights to a set of grades in a course roster to determine final grades, to apply weights representing probabilities to certain measurements you are combining, and other more mundane matters.

- **26.** *Relaxation.* Laplace's equation,  $\nabla^2 V = 0$ , tells us that the electric potential (or the temperature) at any point in free space is the average of the four closest points. Have an  $8 \times 8$  array represent points for which the electrostatic potential is to be determined. The borders of the array are as follows: the top capacitor plate is at 100 volts, the bottom plate is at 40 volts, and the two side plates (each 7 units long) are both at 0 volts. The values of the electric potential at the interior points could be determined by solving simultaneous linear equations (see Chapter 14), or they can be determined by assuming initial values (say, 0) for the interior points, and then calculating their values as the average of the four closest points, performing several iterations on this calculation until the values seem to stabilize. Write a program which will determine the values of the interior points, print out the array, and then ask if another iteration is to be performed.

- 27.** In Chapter 3, Problem 28, we examined a table of PSI (Pollution Standards Index) values and their associated Health Effect Descriptors. There is more detail to be included in such a table (see the Ott reference), such as Air Quality Level and an appropriate Cautionary Statement; these are as follows on the next page.

Write a FORTRAN program using arrays which will accept a PSI value as input, print out the corresponding Health Effect descriptor, ask whether the user wants the Air Quality Level, and then ask whether the user wants the Cautionary Statement printed out (if yes, then do so). It would be efficient to store the three different categories of responses in three arrays and then select the appropriate entry in each case, if needed.

<u>PSI</u>	<u>Air Quality</u>	<u>Health Effect</u>	<u>Cautionary Statement</u>
400–500	Significant Harm	Very Hazardous	All persons should remain indoors; keep all windows and doors closed. Minimize exertion and avoid traffic.
300–400	Emergency	Hazardous	Elderly and those with diseases remain indoors. Everyone should avoid physical activity.
200–300	Warning	Very Unhealthful	Elderly and persons with heart and lung disease remain indoors; no physical activity.
100–200	Alert	Unhealthful	Persons with heart and respiratory ailments remain indoors; reduce physical activity.
50–100	psi % of NAAQS	Moderate	None
0–50	psi % of NAAQS	Good	None

## SUGGESTED READINGS

Amsterdam, Jonathan. "An Analysis of Sorts." *BYTE*, September 1985, 105–12.

Bentley, Jon. "How to Sort," in "Programming Pearls." *CACM* 27, no. 4 (April 1984), 287–91.

Bentley, Jon. "Thanks, Heaps," in "Programming Pearls." *CACM* 28, no. 3 (March 1985), 245–50.

Boothroyd, J. "Shellsort." *CACM* 6, no. 8 (1963), 445.

Edgar, Stacey. *Advanced Problem Solving with FORTRAN 77*. Chicago: SRA, 1989. See Chapter 6.

Hoare, C. A. R. "Algorithm 63: Partition" and "Algorithm 64: Quicksort." *CACM* 4, no. 7 (1961), 321.

Hoare, C. A. R. "Quicksort." *Computer Journal* 5, no. 1 (1961), 10–15.

Knuth, Donald E. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Reading, Mass.: Addison-Wesley, 1973.

Shell, Donald L. "A High Speed Sorting Algorithm." *CACM* 2, no. 7 (1959), 30–32.

# CHAPTER 8



## ERRORS—A FACT OF LIFE

Humans make errors; computers do what humans tell them to do. No matter how long you program or how skilled you become at it, there still will be oversights in your programs that make them fail or give wrong answers. So you must become the best “defensive programmer” you can, on the alert for errors—your own and those made by users of your programs. This chapter alerts you to the most common types of errors, and how you may best protect against them.

Scientists will use computers to recalibrate the optics of the Hubble Space Telescope's sensors. This will compensate for its flawed mirror and allow the telescope to resolve very faint images.

*"A little neglect may breed mischief . . . for want of a nail, the shoe was lost; for want of a shoe, the horse was lost; and for want of a horse the rider was lost."*

- Benjamin Franklin, Poor Richard's Almanac

Once in a great while, you may write a "gold-star" program, one which compiles with no errors on the first try, and which runs and gives reasonable results. However, it is more likely in most cases that you will first have to grapple with *syntax*, or grammatical, errors, which the compiler will detect and, even when those are removed, you may still have logical or *semantic* (meaning) errors which cause your program to give no answers, too few answers, too many answers, the wrong answers, or to get stuck in an infinite loop. You may also encounter execution errors, which cause your program to halt abruptly. All such errors must be dealt with, and we have included some advice here to help you.



## SYNTAX ERRORS

The compiler will catch any instructions which are not properly formed according to the rules of FORTRAN. This sort of problem could occur by having a statement number too far over in the entered line (extending beyond column 5), an instruction beginning before column 7 (unless you are on a "freefield" system which allows this), by not having a single variable on the left of an assignment statement, an unmatched parenthesis, adjacent operators, loops or block IFs whose ranges overlap, a misspelled control word (many people misspell INTEGER, for example), and a variety of other errors. A good diagnostic compiler will give error messages which are pretty helpful, but unfortunately many of the diagnostic routines are relatively oblique in their pronouncements. Since this will vary from compiler to compiler, you will have to learn to live with both the good and the bad.

### Some Common Syntax Errors

Columns in the entry line

123456789...

100 CONTINUE

Problem: Statement number goes beyond column 5

QUEST = QUEST + 2.0

Problem: FORTRAN statement begins before column 7

A-B = 5.0

Problem: Not a single variable name on left of =

KAT = (MAT\*\*4-(IA\*3))

Problem: Unmatched left parenthesis

```
SMALL = 10.0**-6
BIG = 60.0****5
```

Problems: Adjacent operators {though *some* compilers might allow the first, and interpret it as `SMALL=10.0**(-6)` }

```
ISUM = 0
DO 20 I = 1, 10
  IF (I.LE.5) THEN
    ISUM = ISUM + I**2
  ELSE
    ISUM = ISUM + I
20  CONTINUE
ENDIF
```

Problem: DO and Block IF ranges overlap (as indicated)

```
RAEL SUNNY
INTERGER A, B, KATE
```

Problem: Misspelled type words—REAL and INTEGER

When reading diagnostics, take a good hard look at the line flagged by the compiler, and perhaps at the line immediately before it and the line immediately after it, since a stray character in column 6 can accidentally cause one line to be considered a continuation of the line before it, and account for many otherwise incomprehensible errors. If an error message proclaims illegally nested DO loops, and you only have one DO loop, then look for a block IF to see if its range overlaps that of the DO. Draw out the ranges of such loops and IF blocks on a copy of your program listing to check for such overlaps yourself.

Have your text or a manual handy, so that you can make a quick check for the proper form, if necessary. If you try to do a nonstandard operation, such as a WHILE DO, be sure to consult your system manual, since the syntax may be peculiar to your own system (for example, VAX/VMS uses a DO () WHILE form), or may not even exist on your system. In the latter case, substitute the way of simulating such a control structure discussed in Chapter 3. A program error is often referred to as a “bug,” named, according to tradition, after a dead moth that was found in the inner workings of an early computer when the users looked for the source of a persistent error. The more practice you have in “debugging,” the better you will get at it. Errors will begin to jump out at you from the screen or listing, as you become more familiar with properly constructed code.

You should always begin by writing your program on paper or with a good screen editor, in some quiet setting away from distractions. Do not try to compose your program at the terminal without any forethought—this is sure disaster! Enter the carefully designed program to the computer, trying to avoid typing errors such as letter transpositions. A compiler will diagnose improp-

erly formed FORTRAN statements, but there are many other “syntax” errors that the compiler will not catch. For example, if you have transposed the letters in a variable name such as INSANE, so that at another point in the program you have called it INASNE, the compiler will simply treat these as two different, but legal, integer variables.

One way to catch such typos is by very carefully reading over a copy of your program listing, and another is to obtain a symbolic reference map when your program is compiled (this usually just requires an additional parameter specified on the compile command, such as “CROSS-REFERENCE”). Such a reference map will list all of your program variables, in alphabetical order, as well as indicating the program line numbers on which they are referenced (used) and defined (given values, by assignment, DATA, or READ). It is usually easy to spot a mistyped variable name in such a list, especially if it comes up flagged as initially undefined. The reference map will also list all your statement numbers, in numeric order, and the program lines which reference them. Another option that may help with the mistyped variable name problem is to use IMPLICIT NONE, if available on your system (all compilers will have it in Fortran 90)—this then *requires* that you explicitly specify the type of all program variables at the beginning of your program, and any names not so typed will show up as errors (see Chapter 5 and Appendix E).

If you get many compiler errors, and most of them seem incomprehensible to you even after reviewing the syntax rules, you can get the system to “list” them all out on paper for you along with the program listing. This simply requires a command (such as “LIST”) along with the compile command, that will be specific to your particular system. Find out what it is on your system. You can then take a “hard copy” of your program and its problems to your instructor or someone at the computer center for diagnosis and probable cure.

Other common errors that may occur in typing, and which may not cause compiler diagnostics, are mistaken substitutions of I for 1, or a lower-case letter ‘l’ for a 1 (if your FORTRAN system allows lower-case letters), or the confusion of a letter ‘O’ for a zero (0); watch carefully for such errors. You can usually tell the zeros from the letter ‘O’s on a printout or the screen by comparing them to zeros in statement numbers or constants. Another error that the compiler will overlook is two (short) variable names written next to one another, in the mistaken belief that this will imply multiplication, as it does in algebra. If you write:

$$C = A B$$

expecting it to multiply A by B and store the result in C, you will be sadly mistaken. FORTRAN ignores blanks (except in literal strings within quotes), and so would interpret this as:

$$C = AB$$

and assign C the value of some (probably undefined) real variable called AB.

Such subtle errors are hard to unearth, but a look at the symbolic reference map (or use of IMPLICIT NONE, if available) will bring such problems to light.

Certain expressions cannot be combined with expressions of other types. The strict rules against “mixed-mode” expressions enforced in early FORTRAN compilers have been relaxed, and reals can always be mixed with integers (with the real dominating, and making the result real), or reals or integers mixed with double precision values (in which case the integer or real is converted to double precision) or with complex values (in which case the integer or real becomes the real part of a complex value with a zero imaginary part). However, other combinations of expressions are forbidden. For example, it seems obvious that you cannot modify a character value by *adding* it to a numeric value, or *multiplying* it by a numeric value. Further, combinations of double precision and complex expressions are prohibited in FORTRAN 77. Thus, any attempt to combine constants, variables, or expressions of incompatible types will result in a syntax error.

All syntax errors must be removed before your program can be run, and at that point a new, more difficult stage of debugging begins. A program may be syntactically perfect, yet contain many logical errors, just as an English sentence may be grammatically correct but still convey nothing, or nonsense.

The best way to avoid many errors, syntactic or semantic, is to write your program carefully in the first place. Write it out when you are unhurried and clearheaded, with your text or manual handy, and use a pencil with a large eraser (or else a terminal with a good editor). Attack small segments of your program as separate units, and try to get each one correct before going on to the next. Use meaningful variable names, since these will help you maintain coherence as you proceed. Take advantage of the structured loops and conditional blocks that are available in FORTRAN 77, since they allow the expression of clear, readable code. The more programs you write, the more easily good code will become second nature to you. Practice, practice, practice.



## EXECUTION-TIME ERRORS

The most blatant execution-time error is an infinite loop. Be sure that you know the system command to interrupt or stop program execution, so that when such a loop occurs you do not sit around helplessly while the program runs away uncontrolled. Once you have stopped the program, the system should display some information regarding the line at which the program was interrupted. Since this line should be part of the offending loop, look carefully at the line and the context in which it occurs. You probably will not have a simple, conspicuous *error* such as:

10 GO TO 10

and many compilers would have recognized that as a problem statement to begin with. Look carefully at the repetition structure you have set up, and

critically examine its exit conditions—can they be met, given the way the loop is set up? Do they depend on values input from outside the program? Could those values have been entered incorrectly?

Are your loop parameters all of the same type? If you have an integer loop variable and a real step size, this could cause problems, as in the following example:

```
DO 20 INT = 1, 5, 0.5           {causes infinite loop!}
```

since each time the step size is added to INT, the result is truncated, and the upper limit is never reached. FORTRAN 77 pre-checks DO loops, so if you accidentally wrote an impossible loop:

```
DO 70 JAKE = 3, 10, -1
```

this loop would simply be skipped (but care should be taken not to create such situations, or to rely too much on the pre-check). Loop parameters should only be set on the entry DO line, and never changed through the range of the loop. Do you alter any of the parameters in the loop? This can cause unpredictable damage (some compilers might catch this and give a syntax error message, but others may overlook it).

There are other ways to construct loops besides using DO's, as we have seen. You can simulate a DO, a WHILE, or a REPEAT/ UNTIL using IFs and GO TOs. In these situations, you must be careful to see that the value or values controlling the looping condition are changed during the course of the loop, so that at some point the terminating condition can be met.

Another error you may encounter is that your program runs, comes to a STOP, but never gives you any answers. The most obvious cause of this could be that you neglected to include a PRINT or WRITE statement for the result, or that this statement is never reached. Look for the output statement, and the path(s) to it, to be sure it will be executed. Again, a symbolic reference map may be helpful in tracking down the paths to such a statement, and determining if they are followed.

If you are getting *too many* answers, your output statement is probably in the wrong place. Look carefully at its position in the program, and determine where it should be placed. Wrong positioning can also be the cause of too few answers, or answers output before the operation is completed.

A run-time error that will cause program termination is that of division by zero. Look carefully at the operations involved on the line that is identified, and see if you can track down how the divisor expression could have gone to zero. Perhaps you just need a test on the divisor prior to the calculation, which skips the calculation if the value is zero, or perhaps there is some other program error that causes the divisor to become zero when it should not be so. Have you written the FORTRAN equivalent of the problem equation correctly? Are all variables involved in the expression getting their values correctly before this point?

It is essential to initialize your variables. Do not assume that the machine you are on will have “zeroed memory” between programs, so that any locations you have not given values to will be zero. This is done on some machines, but not on all. Even if you know your machine does so and use that fact, your program will not be portable to other computers. Some machines simply do not clear memory between programs, so that an uninitialized variable may contain some value (or instruction!) from a previous program. Other machines set all uninitialized locations to an illegal value, so that if you try to use them without giving them values yourself first, an execution error will occur.

Your calculations may get out of hand due to some mistake in a formula, or in a loop limit, and create an *overflow* or an *underflow* condition, which can cause program termination. The word size on the computer you are using, and the way in which a location containing a real value is divided up for representation of exponent and mantissa, determine certain limits on the maximum size an integer or real value can have. This was discussed briefly in the Introduction, where we saw that a computer with a 32-bit word, such as an IBM or a VAX, will have a limit of 2147483647 as the largest integer it can accommodate in one location, and even a machine with a large 64-bit word, such as a Cray, cannot handle integers larger than 9223372036854775807. Thus an attempt to compute an integer value that exceeded these limits would cause program termination due to an *overflow* error.

Similarly, as we discussed in Chapter 2, a machine will have limits on the magnitude of a single-precision real it can store. On a 32-bit machine this can vary—for example, on an IBM it is on the order of  $10^{75}$ , but on a VAX it is on the order of  $10^{38}$ . Even on a 64-bit machine such as a Cray, the largest single-precision real is on the order of  $10^{2466}$ . Thus, calculated values that would exceed these limits cause errors on these machines.

An *underflow* will occur if a real gets *too small* for values that can be accommodated in a word on your machine. For example, on a 32-bit IBM, the smallest real is on the order of  $10^{-75}$ , on a VAX it is on the order of  $10^{-38}$ , and on a Cray, it is on the order of  $10^{-2466}$ . Thus a computed value smaller than these limits will either cause an underflow error condition, or be set to zero. These machines have some practical built-in limit that defines what an essentially “infinite” value is for that machine (a value too large for it to store), or an essentially “infinitesimal” value (one too small to be represented).

Input or output of a value with a format descriptor of the wrong type will also create an execution error. Be sure that you use integer formats for integer values; F, E, or G formats for reals; A for characters; and so on. As we have seen earlier, an attempt to output a numeric value in a field that is too small to contain it will result in an output containing all stars (\*) in the defined field width. This will not cause premature termination of your program, but it could result in a nasty error message from some systems in the middle of program execution.

Another run-time terminating error can be caused by an array subscript that goes out of range. Some systems allow this to happen, and then the

damage will show up elsewhere in your program, and be even more difficult to detect. If it is not detected, it can cause serious “overwrite” of portions of your program or other variables by the array values. However, you may get an “invalid index” or “illegal access” or similar error message if your system does not allow subscripts to go out of bounds. If you get such a message, look at the line flagged and carefully examine any array subscript expressions. Trace back the logic that determined these values, and see how they could have acquired values that are either too high or too low for the array bounds specified by your dimension statement. Perhaps the array bounds only need to be expanded, but it is more likely that your logic in calculating the subscripts is erroneous. An example of such a program error would be the following:

```
REAL A(20)
DO 50 K = 1, 50
    A(K) = 2.0**(1-K)
50 CONTINUE
```

In this example, either the loop limit should be set at 20 or else the array A should be dimensioned to 50.



## LOGIC ERRORS

The execution-time errors just discussed are probably due to errors in your program logic, such as missing a step, or making an increment at the wrong time. However, many logic errors will not have results that cause some computer error condition that will abruptly terminate your program; they are subtler and more insidious. An error of this type is best prevented by careful problem analysis and program coding. Next to that, it is best to lay “traps” that will detect such errors. One way to do this is to run very simplified “test” data through your program, data for which it is easy to hand-check the answers. However, do not hand-check only the very simplest values, or you may not be giving your code a thorough test. Try simplest-case values, and then some slightly more complex values. Then, if your program does not give the correct answers, you must dig deeper to unearth the problem. If your hand-check had several intermediate stages in the calculation, insert extra PRINT statements in your program code to print out the intermediate results at those various stages; this will allow you to further isolate the problem.

Hand-checking your program is always a stage you should go through, even before you submit the program to be compiled or run. You should “play computer” and literally follow all the instructions you have written, keeping track, in a table, of the different values your variables take on as a result of executing the instructions. A careful check like this should make clear any places where your code does not do what you had intended it to do.

It is very important to learn how to "trace" through the logic of a program and determine how the values of important variables change, whether in a program of your own you are trying to perfect, or that of someone else which you are trying to understand, and perhaps correct or update. We suggest setting up a table of the values of important variables, and then filling it in as the program execution progresses. You were given some exercises like these at the ends of Chapters 3 and 4; let us look at such problems again for the purposes of tracing. For example, imagine that you were given the following code to analyze, and were asked what it would print out, or at least what would be stored, at the end of the program. Set up a table, as at right, for the variables involved, and change the entries in the table as the program changes the values of the variables.

INTEGER A, B	A	B
A = 7		
B = 9	7	9
A = A + B	16	7
B = A - B		
A = A - B	9	

We thus see, by tracing the values of the variables, that this code interchanges the values stored in locations A and B, without the use of a third, temporary location. Try the following:

INTEGER ONE, TWO, N	N	ONE	TWO
READ*, N			
ONE = 0	3	0	0
TWO = 0		1	1
DO 40 I = 1, N			
ONE = ONE + I	3		9
TWO = TWO + I**3		6	36
40 CONTINUE	5	0	0
PRINT*, N, ONE, TWO		1	1
		3	9
		6	36
		10	100
		15	225

Since the preceding program did not specify a value for N, we tried out two different (but rather small) values in our table.

Notice that you could also compare your "hand-check" with what the program actually does by inserting a PRINT\*, ONE, TWO statement in the

program right before statement 40. Some interesting observations can be made on the basis of these two traces. As we would expect, the loop running from 1 to 5 has the results of the loop running from 1 to 3 as a subset (which made the initial evaluation of the second table easier). The location ONE sums up the integers from 1 to N, and the location TWO sums up the cubes of the integers from 1 to N. A careful look at our results will reveal that TWO is, at every point, equal to the square of ONE. We would also have seen this if we examined the analytical formulas (instead of loops) for computing these sums:

$$\sum_{i=1}^n i = n(n+1)/2 \quad \sum_{i=1}^n i^3 = n^2(n+1)^2/4$$

In setting up loops, very often the programmer may make "off-by-one" errors. This is less likely to occur in a straightforward DO loop, where it is clear that it will process all values of the loop index from the initial value through the final value. However, if you are setting up your own loops using IFs and GO TOs, for instance, in cases where you are simulating WHILE loops or REPEAT/UNTIL loops, look very carefully at your test for termination, or have the values of the variables which control loop repetitions printed out when they get close to the terminal value. Very often you will find that, for instance, a "less than" test should have been a .LE. test, or something similar. These errors cause your loops to be "off-by-one," executing one time too few or one time too many.

Similarly, in a sort procedure such as the "bubble sort," if you are sorting into ascending order and make the test for correct order (no changes required) that the value examined should be *less than* the value below it, then your program may seem to run correctly for months, until data containing duplicate values is submitted (at which time your supposedly debugged program will spin into an infinite loop). The test should be *less than or equal to*.

Try to consider special cases that may cause problems. You may not think of them all at first, but just the effort of trying to think of them will be productive—later, if not immediately. Problem cases often occur at "boundary values," the extremities of your loop counter, or values lying just inside or outside of specified allowable ranges, such as array bounds or limiting values you test for on input data. Be sure your program behaves as you want it to for such boundary values, and does not "blow up" if a value out of the legal range is generated or input. As we pointed out, a common problem in sorting routines is not properly handling equal values in the array; be sure you carefully check how your code behaves in cases of equal data values.

Include extra PRINT statements at strategic points in your program to output intermediate values, so you can check that they are progressing correctly. Such statements can easily be removed later, or made into Comments or made conditional on the value of some logical "debug" parameter, such as IF (DEBUG) PRINT ... It is often wise to leave them in conditionally, in case you need them in the future when some unsuspected bug shows up.

Have the variable DEBUG set to .FALSE. in the program for normal execution, but allow the user to input a command which can change the value of DEBUG to .TRUE. if the extra print statements should be activated to track down some new error. Just do not place them in tight loops, so that you get much more output data than you can possibly examine. You might want to set up such PRINTs to execute for the first N values, or every M times, for example.



## DEFENSIVE PROGRAMMING

Write your code carefully, with the text or manual handy to check the syntax of any commands you are not sure of. Write and debug your code in logical sections, so that the segments can be carefully checked before they are incorporated into the overall whole. Do not have GO TO statements jumping all over the place, or neither you or anyone else will be able to follow your program. Write logical, well-constructed sets of instructions. Make use of good control structures such as block IFs wherever appropriate. In designing your program, make use of program "stubs" (incomplete program sections that may just print out a message that indicates they have been reached) for the parts that have not yet been developed, so that you can at least see how all the program sections interconnect.

The first person you have to "defend" against is yourself. Just as in defensive driving, the first important thing is for you to be a good driver, so you must be a good, careful, logical programmer. If you are, many potential sources of error will have been bypassed to begin with. Next, you can worry about the "other guy." If anyone is going to use your program besides yourself (which is likely in any environment outside of the classroom), you will have to make very sure that the user(s) cannot bring your program to its knees by bad input data. Thus an important part of defensive programming in any production program is to make it "robust"—that is, able to recover gracefully from all sorts of abuse and incorrect data. This generally involves use of special tests on input data to be sure it is in legal range, and perhaps the use of the ERR= clause (to be discussed in exercise 15 at the end of the chapter, and in Chapter 11) to allow you to reprompt the user for correct input rather than crashing your program. "Echo" any input values by printing them out, so that the user can check them. All of these precautions will also make your program much more accessible to members of a programming team, should you be working in such a group.

Try to keep any input required from the user as simple as possible; do not demand too much of an unskilled person who is probably unfamiliar with computers, or at least with FORTRAN. Show examples of the way the data should look. If possible, in most cases it may be simpler to read the users' input data with a list-directed READ\*, since this only requires that they separate input values by blanks or commas. However, they will then have to enter

character data (such as a 'YES' or 'NO' response to a question) enclosed in single quotes, so explain that carefully.

Expect that you will make errors as you program. Try as best you can to anticipate them, or to provide traps that will bring them to light. Expect far less of those who will use your program. Expect them to make outrageous errors, to hit RETURN in a panic in an interactive program, or whatever the worst case is you can imagine. Try to write your program so that it can withstand these outrages.

Make use of the PARAMETER statement to define certain important constants in your program by giving them symbolic names. Not only does this make it easier for you to change such values at all of the places they occur in the program should the need arise, it also makes it impossible to enter such a constant incorrectly at one place in the program by transposing digits. Also be aware of the fact that each program unit (main program and subprograms) needs its *own* PARAMETER statement. A symbolic constant defined in the main program will not automatically carry over into the subprograms.

Rereading your program is a good idea, since you may find errors or ways to make it more elegant or efficient. However, just as in writing prose, you can be too familiar with your own work and overlook errors because you know what you wanted the effect to be. Thus if you can get a competent friend to read over your work, you will get the advantage of a more objective viewpoint. This principle is incorporated in industry in the "structured walkthrough," in which a number of colleagues will subject your code or overall program design to scrutiny. It has proved to be very productive. It does, however, require that the programmer not have too much ego wrapped up in the program.

Just as a teacher really learns the course material best when pressed to teach it to a class, you will understand your program best (and perhaps uncover previously hidden errors) by trying to explain it to someone else. This person could be your competent friend, or even someone relatively unfamiliar with computers—perhaps the sort of person who might have to use your program later on. You can gain insight into the difficulties such users might have, or bring down upon the program, by talking to them early, rather than after the damage has been done. On your coding sheet or program listing, carefully draw the ranges of *all* DO loops, simulated loops, and block IF structures. This will help you ensure that these ranges do not overlap. Even though your compiler may allow a DO/END DO structure, without the need for a statement number on the terminal statement of the DO, you may find that using such statement numbers helps you to identify the ranges of your DO loops more easily.



## ROUNDOFF ERRORS

There are limitations, as we discussed earlier, on the size of integer or real values you can store on your machine. These limits will vary according to the computer

you are using and the word size it has, but you should be aware of the limitations. A 32-bit word size is common, and will handle integers up to a magnitude of 2147483647. Such a machine may be able to store real numbers on the order of about  $10^{75}$ , to about six decimal digits precision (for example, some IBM machines), or on the order of  $10^{38}$ , to about seven significant decimal places (a VAX, for example). If the integer values you must deal with are larger than the word capacity of the machine, you will have to use an array of locations to store the digits of large integers. If your real values need greater magnitude, or more significant digits, you will have to use DOUBLE PRECISION values, which will use two memory words to store each real value.

Even with double precision values, there still is a limit on the number of significant digits that can be stored. This can be readily seen for repeating decimal fractions like  $1./3.$ , which can never be stored precisely in a finite decimal expansion. Thus, the *precision* of my result will vary greatly depending on whether I calculate with a value of  $1./3.$  good to three places (0.333) or good to 6 places (0.333333). Unfortunately, on a *binary* machine of the kind you are using, even most simple-looking decimal fractions like 0.4 cannot be stored precisely, since they have an infinite *binary* expansion. For example, the binary equivalent for 0.4 in decimal is 0.01100110..., infinitely repeating (this may also be written as  $0.\overline{0110}$ ). Thus there will be some precision lost in storing such values, and if you add the binary representation of 0.4 to itself five times, you will probably not get 2.0, but rather something like 1.9998. This is because what is actually stored, when the infinite "tail" of digits in the expansion is lopped off, is something *less than* 0.4 (just as, analogously, 0.333 is less than  $1./3.$ ). This is called "roundoff error," and you should be aware that it exists. Sometimes you can attempt to compensate for it, and at other times it simply means you have to perform tests differently (for example, testing whether two real values satisfy the condition of being *less than or equal to* each other, rather than testing for exact equality of the values).

## Propagation of Errors

If there is an error in representing one value (either due to roundoff error, or inaccuracy of measurement, or an imprecise formula), then this error can be *propagated* if this value and other values of similar imprecision are combined in arithmetic operations. For example, if our machine can represent fractional values good to six decimal places, then each has a potential error on the order of  $10^{-6}$ , or at least  $5 \times 10^{-7}$ , due to roundoff error. If  $n$  of these values are added together, the error increases to  $n \times 10^{-6}$ , and if a particular value is multiplied by  $m$ , the error is magnified  $m$  times; this is referred to as *propagation of errors*. As an intelligent programmer using a computer with such inherent limitations, you must be aware of the effect these limitations can have on the goodness of your results.

## Cancellation Errors

Another difficulty is that if a large real value is added to a very small value, the small value may have little or no impact on the large value, because of the number of significant digits available in the result. As an analogy in decimal, if you add 4.5 E 4 (45000.) and .33 E -3 (0.00033), and you can only store 6 or 7 decimal digits of the result, the result will be 4.5 E 4, thus showing *no* effect of adding the smaller value. The larger value has “cancelled out” the effect of the value of much smaller magnitude. Thus, if you have a long sequence of terms—in decreasing order—to add up, it may be advantageous to add them in reverse order, smallest terms first, instead of the other way. This way the smaller terms are added to each other, and so taken together may amount to enough to make an impact on the larger terms. You can experiment with this, for example, by adding the sequence:

$$1 + 1/2 + 1/3 + 1/4 + \dots 1/10000$$

in both directions, and comparing the results you get. Which result is more accurate?



## SYMBOLIC DEBUGGERS

Many modern computer systems incorporate a useful tool for the programmer called a *symbolic debugger*, or simply a *debugger*. Usually such a tool is activated by a command such as DEBUG. It is used to help isolate execution-time errors and logic errors in a program, and correct them. Compile-time errors still must be handled by the painstaking method of trying to decipher the cryptic messages provided by the compiler’s diagnostic system.

A *symbolic debugger* allows the programmer to observe the program operation closely, and to make changes and see how these changes affect the problem conditions. Any element of the program which is identified by a symbol—a variable, a labelled line, or a subprogram—can be monitored by the debugger. The programmer can start and stop program execution at will, and ask the system to *step* through a section one instruction at a time. The contents of flagged variables can be examined as the program proceeds, or execution can be set to stop whenever a particular variable is changed, so that it can be inspected. The programmer can use the debugger to change the type or the value of a variable, or may actually allow the programmer to change instructions in the program, and observe the effects, without the inconvenience of having to go through the compile/link/execute routine every time an alteration is made. Such flexibility greatly facilitates the finding and correction of a wide variety of program errors.

Consult your computer center personnel or the system manuals to determine if a symbolic debugger is available for you to use.



## SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

*Syntax errors* are those that are caught by the compiler. They are violations of the grammatical rules of FORTRAN, such as unmatched parentheses or misspelled key words. The diagnostic message may be helpful in finding the error, but if it is not, scrutinize the flagged line carefully. Make use of symbolic reference maps or IMPLICIT NONE to force typing of all variables, if available.

*Execution-time errors* are those which cause some violation recognized by the monitor, such as an infinite loop, overflow or underflow, invalid array subscripts, or division by zero. In such cases, the offending line will probably be flagged, and you can look for the error.

*Logic errors* are those which are most difficult to find. They are not caught by the compiler and do not cause premature program termination, but nevertheless cause the program to go awry, to give wrong answers, etc. Careful "hand-checking" of your program logic, and tracing the values of crucial variables, will help avoid such problems. Extra PRINT statements in your program during development stages help find such errors.

Program "defensively," by writing careful programs and trying to anticipate errors an unskilled user of the program may make. Use the PARAMETER statement to define important symbolic constants. Be aware of the difficulties caused by roundoff errors and differing precision on different machines. If your system has a symbolic debugger, learn how to use it.



## EXERCISES

1. Be sure that you know how to stop your program when it goes into an infinite loop. This may involve using a "break" key or some special *control* operation (usually involving holding down the CONTROL (or CTRL) key at the same time as a special key to initiate program interruption). Deliberately create a program with an infinite loop in it, and see what information you get when you terminate the program in the middle of the loop. You can probably find out at least what program line you stopped at, which helps you find the locus of the infinite loop. Since you can now stop the program almost at will, include a print statement which will reflect the contents of the important loop parameters, so that you can see how the way they change (or fail to change) affects the execution of the loop.

Use these newly found skills to find the source of the infinite loop in the following example (since type statements are not included, default typing is in effect):

```
SUM = 0.
DO 20 N = 1, 10, 0.5
    SUM = SUM + N
20 CONTINUE
PRINT*, SUM
```

2. Determine if your FORTRAN compiler has the IMPLICIT NONE feature. If it does, make use of it in a program, and declare all of your variables, but misspell one in the body of the program. See what sort of error message the compiler gives you.
3. Create a program in which an IF block range overlaps the range of a DO loop, and see what sort of error message you get. An example of this error would be the following program:

```
INTEGER TOTAL
TOTAL = 0
DO 50 N = 1, 10
    IF (MOD(N,2).EQ.0) THEN
        TOTAL = TOTAL - N
    ELSE
        TOTAL = TOTAL + N
50 CONTINUE
ENDIF
```

Familiarizing yourself with the error messages you can expect in certain situations will greatly aid you in knowing the probable cause when those error messages crop up in an actual program.

4. Create a simple array, dimensioned to some small number of locations. First try an overt reference to a subscript out of the range of that specified for the dimension, and see if the compiler catches it, and what message it gives you. For example, you might try:

```
INTEGER KAT(10)
KAT(15) = 15
```

Now create a variable subscript for your array, so that the problem cannot be caught by the compiler. For example,

```
INTEGER KAT(10)
N = 15
KAT(N) = N
```

Try running the program, and see what sort of run-time error this creates. It may refer to "Invalid access," or some such message.

5. Another sort of "invalid access" is a call to a subprogram with the wrong variable type. Try out a call to a system function (one that is not generic), using the wrong type of argument. For example, try

```
REAL A
INTEGER N
DATA A, N /3.0, 4/
PRINT*, FLOAT(A)
PRINT*, IABS(A)
R = ALOG(N)
```

and see what this causes.

Now try a simple defined function of your own (to be discussed in the upcoming chapter) and give it the wrong type of argument, to create a similar error message. For example,

```
PRINT*, MYMOD(25.0, 10.0)
STOP
END
FUNCTION MYMOD (M, N)
MYMOD = M - M/N*N
RETURN
END
```

6. Another subtle error that can occur with your own defined functions is that of having them treated as one type in the calling program and as a different type in the function itself. See what sort of mischief the following example will cause on your system.

```
PRINT*, HYP(3.0, 4.0)
STOP
END

INTEGER FUNCTION HYP (A, B)
HYP = SQRT (A**2 + B**2)
RETURN
END
```

Then try the same example, but reverse the typing problem—make it integer in the main program and real in the function, by adding the line INTEGER HYP at the very beginning of the program, and removing the word INTEGER where it occurs before FUNCTION HYP; see what sorts of difficulties this causes. The

function, as defined, should calculate the hypoteneuse of any right-angled triangle of short sides A and B. Thus, given the arguments 3.0 and 4.0, it should return a 5.0 or a 5, depending on the type of the function itself. Does it? If not, why not?

7. What compiler syntax errors are created by improprieties (adjacent operators, unmatched parentheses, and various other expressions that are not well-formed) such as the following:

```
NONE = 3*-2
HELP = (((A**B + (C*D)))
A + B = B + A
```

Create some of your own, and test the results. Get to know your compiler and its error messages well. It will pay off!

8. How does FORTRAN interpret the following instruction? Write a short program to check it out, and use your developing skills to isolate and fix the problem (perhaps get a reference map to determine what variables are being used).

```
N = -5
IF (N.LT.0) THEN N = -N
PRINT*, N
```

What do you think the person who wrote such code intended to do? What problem is there, and how do you fix it?

9. Try a short program in which you use an IF/THEN instruction, but omit the matching ENDIF (an error commonly made by beginning programmers), and see what error message from the compiler it creates. For example,

```
READ*, A
IF (A.GT.90) THEN
PRINT*, 'GOOD!'
ELSE
PRINT*, 'YOU GOOFED'
B = B + A
```

10. What does your system do with division by zero? First try out an explicit error, one that the compiler should catch:

```
A = 100./0.
```

Then try a more subtle approach:

```
N = 8
M = N - N
PRINT*, N/M
```

11. What does your system do with uninitialized variables? Some systems zero memory (so that any uninitialized variable is set to zero); others simply leave such locations as whatever they were in the previous program's use of the space; and others set any undefined locations to some *illegal* value, which will then cause a run-time error if you attempt to use these locations without having first given them a value. See what method your system employs. For example, you might try:

```
DO 7 N = 1, 5
7      S = S + N
```

without first giving any value to S. What happens?

*Note:* Even if your system does zero memory, it is dangerous to come to rely on this, and not initialize your variables. Such habits will make your programs nonportable to other computers, which may use some other approach to uninitialized variables.

12. A useful tool in debugging complex programs is the NAMELIST feature, which is not part of the 1978 FORTRAN 77 Standard but is available on many systems. Check to see if it is available on your system. If so, you will find it a handy way to label a whole group of otherwise unrelated variables with a single name, and then put in a debugging PRINT statement that simply asks for the group by name. In addition, all of the variables in the group will be printed out with their names, so that you have a conveniently labelled output to read.

To use NAMELIST, if available, you include a NAMELIST statement in the declaration part of your program, of the form:

```
NAMELIST /name1/list1 [ [,]/name2/list2 . . .]
```

The names given to NAMELIST groups must follow the FORTRAN variable naming conventions. A variable may occur in more than one NAMELIST group. No formats need to be used when printing out a NAMELIST group. Thus an example of its use would be:

```
INTEGER B(4)
NAMELIST /BUGS/ A, N, MAX, B, /BUNNY/ X, Y, COUNT
```

and the output statement would then be of the form:

```
PRINT BUGS { or PRINT BUNNY}
```

which would output, for example:

```
&BUGS A = 6.600000, N = 5, MAX = 55,
B(1) = 9.900000, B(2) = 4.333333, B(3) = 2.211111,
B(4) = 8.666667 &END
```

The output line begins with a space, followed by an ampersand (&) (or, on some systems, a \$), followed immediately by the group name, followed by the list of variables in the group and their respective values. All values in an array will be displayed, each with its appropriate subscript. The list of variables and their values is terminated by &END (or \$END on some systems). Real values will be output with as many places precision as are available on the system in use.

This is a simple, handy way to output a whole group of variables, each with its name indicated as well as its value. You can readily see its use in debugging a program. Try it, if it is available to you. You can also use NAMELIST for input, in much the same manner. The input record must look much like our output record example, beginning with a blank followed by an & (or \$) followed immediately by the group name, then the list of variables and their values, and terminated by &END (or \$END). A NAMELIST output record written out to tape or disk, or punched on a number of cards, could then be used as NAMELIST input.

*Note:* NAMELIST will be available in Fortran 90.

- **13.** Try to create overflow and underflow conditions on your computer. See what error message they produce. Does underflow create an error message, or simply go to zero? To create an overflow, for example, you can look up the value of the largest integer allowed on your system (for example, if your computer has a 32-bit word, it will be 2147483647), and try assigning or computing a value larger than that. Another way is to set up a loop that will calculate (and print out) powers of two (or 10, if you prefer) until an overflow occurs. Try to determine the last legitimate value before the overflow occurred, and then try proceeding by adding 1s until the overflow occurs again. In this way, you can determine precisely where the cutoff point occurs.

For underflow, proceed in a similar manner, except this time keep dividing by 2 (or 10) until underflow occurs. Note the last legitimate allowed value, and proceed from there.

- **14.** At the end of the previous chapter on arrays, you were asked to write a program to determine how many days old anyone is, given a birth date and a current date. Think about how you would go about finding the errors in such a program that gives results that are wrong in one of the following ways:

- (a) person is one day too old
- (b) person is one year too old
- (c) person is one year too young
- (d) the age calculated is too low by the number of days in the current month
- (e) leap years do not seem to be handled correctly

**15.** A run-time error that commonly occurs is trying to input (or output) a variable with the wrong type format descriptor. This usually results in a *fatal* error, which terminates the program. However, one can use a parameter in the READ statement or WRITE statement which will branch to a statement in the program if an I/O error occurs; it is a clause of the form ERR = st#. Thus, such I/O statements might be:

READ (5,88, ERR = 99) list

or

WRITE (6, 66, ERR = 100) list

Make use of such an I/O statement in your program and deliberately create an I/O error that will utilize the ERR = branch; then see if you can figure out how to have your program recover from this error and go on about its work. Note that *any* input line can be processed as a string of characters.

**16.** Compare the precision you get by adding up the terms in the sequence:

$$1 + 1/2 + 1/4 + 1/8 + \dots$$

for 100 terms; for 500 terms

- (a) using single precision
- (b) using double precision
- (c) adding the terms left to right
- (d) adding the terms right to left

**17.** Most decimal fractions (except those such as 1/2, 1/4, 1/8, 3/16, etc.) do not have precise binary representations on the computer. Thus, paychecks may end up being written out which are low by a penny. Write a program which will correct this problem, by calculating the gross pay of an employee who works N hours at RATE dollars per hour. Input N and RATE, and make RATE something not precisely representable in binary, such as \$5.40 an hour. Figure out how to do your calculations without *any* loss of precision, and write out an accurate gross pay amount.

*Hint:* Binary representations of *integer* values are always precise, as long as they are within the representable range of values that will not cause overflow.

- **18.** Many values you calculate with may actually be decimal *fractions*, such as 1/3, 1/4, 1/5, and so on. Most of these are not precisely representable in binary. However, if you could do all calculations using the rational (that is, ratios of integers) notation and output a result as a fraction, it would be correct.

Write FORTRAN program segments which will add, subtract, multiply, and divide any two values, both expressed as the ratios of integers (thus your routines will actually accept *four* integer values, two to define each rational). The result of your operation should also be rational, that is, be given as two integer values, expressed as a ratio.

**19.** Determine if your system has a symbolic debugger. If it does, learn how to use it effectively.

**20.** Write a program to test the effect of propagation of errors. Take a rational fraction  $1/n$  (for instance,  $1/7$ ), and add it to itself  $n$  (e.g., 7) times, assuming (that is, forcing) two significant decimal digits, three, four, up to the number of significant digits representable on your system. Do this by truncating digits after the number allowable for each case. You can do this by multiplying the fractional value by  $10^d$ , storing the result in an integer location, then dividing again by a real  $10^d$ , to obtain  $d$  significant digits. See if you can arrange to have the last significant digit rounded up if the first following digit truncated is 5 or greater, rounded down otherwise. Compare your results with different numbers of significant digits used.

**21.** Your physics professor, Dr. Know, is testing out a new hypothesis he has concocted—that the height ( $r$ ) to which a new substance will rebound is proportional to the square root of the height ( $h$ ) from which it is dropped:

$$r = 1.2 \sqrt{h}$$

Your job is to take  $N$  measurements (say, 100) of rebound heights ( $r'$ ) for various initial heights ( $h$ ) from which the substance is dropped, input  $N$  pairs of values ( $r', h$ ) to your computer program, and have it compute a measure of the variation of your measured values from those predicted by Dr. Know's formula; to calculate the variation, determine

$$\frac{\sum_{i=1}^N \frac{|r'_i - r_i|}{r_i}}{N}$$

Print out the variation measure calculated by your program.



## SUGGESTED READINGS

Beizer, Boris. *Software Testing Techniques*. New York: Van Nostrand Reinhold, 1990.

Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering*. Reading, Mass.: Addison Wesley, 1979.

- DeMillo, R. A., W. M. McCracken, R. Martin, and J. F. Passafiume. *Software Testing and Evaluation: A Report*. Redwood City, Calif.: Benjamin-Cummings, 1987.
- Myers, Glenford J. *The Art of Software Testing*. New York: John Wiley & Sons, 1979.  
A wealth of information on testing techniques, walkthroughs, error checklists, and test-case design.
- Ould, Matryn, and Charles Unwin (eds.). *Testing in Software Development*. New York: Cambridge University Press, 1987.
- Parrington, Norman, and Mac Roper. *Understanding Software Testing*. Englewood Cliffs, N.J.: Prentice-Hall, 1989.
- Van Tassel, D. *Program Style, Design, Efficiency, Debugging, and Testing*. Englewood Cliffs, N.J.: Prentice-Hall, 1977.
- Weinberg, Gerald M. *The Psychology of Computer Programming*. New York: Van Nostrand, 1971. A classic.
- Workshop on Software Testing, Verification, and Analysis, 1988, Proceedings*. IEEE Computer Society Press, 1988.
- Yourdon, Edward. *Managing the System Life Cycle*. New York: Yourdon Press, 1982.
- Yourdon, Edward. *Writings of the Revolution: Selected Readings on Software Engineering*. New York: Yourdon Press, 1982.
- Also see additional references on software engineering listed at the end of Chapter 15.

# CHAPTER 9



# SUBPROGRAMS

Interesting scientific and engineering problems are often complex. It is desirable to be able to break up such a problem into its functional parts, and solve one part at a time. FORTRAN provides the programmer with the subroutine structure, to write independent code to solve one piece of the puzzle at a time, and the function subprogram, which allows the implementation of scientific/mathematical functions. These tools make possible the rational management of a complicated problem that would otherwise present an almost impossible task to the programmer, if it had to be handled in one piece.

Opening a Russian Matryoshka figure reveals the smaller figures contained within.

*"The mightiest rivers lose their force when split up into several streams."*

- Ovid, Love's Cure

We have already seen that it is best to develop your program in small, manageable segments that can be thoroughly tested. Such segments can be seen clearly as functional parts of the overall program. They should not be too long, and should do only one or two primary jobs and be easy to comprehend. A highly recommended method of program design is the "top-down" approach: after the overall structure of the program is outlined, it is broken down into manageable subtasks. The outline (much like an outline for a term paper, or a proposed table of contents for a book) shows the overall skeleton of the program and the ways in which the parts interconnect. The details can be filled in later. These "parts" may be merely segments of code, which are later strung together to form the whole program, but often they can be designed as program units in their own right. You will see the advantages of the use of subprograms in FORTRAN—they provide functional units that can be designed and tested separately from the overall program and later built into a coherent whole.

Subprograms have advantages besides those of letting you isolate and develop program segments. The use of subprograms also allows you to define general-purpose operations which can be used over and over, either in the one program you are writing today or in many programs over the years. Thus, it is very important to study the design of such program segments and how they can be linked to the overall program structure.

We will begin with a simple tool for problem abstraction and generality—the statement function. We will then briefly examine system functions standardly provided by the FORTRAN compiler, and finally branch into developing subprogram units of our own—subroutines and functions.



## STATEMENT FUNCTIONS

A statement function allows you to take a simple operation, one that can be defined in a single FORTRAN expression, and generalize it so that it can be referenced many times and given different values to work on. It lacks some of the flexibility of true subprograms, since it cannot involve loops, arrays, or anything more complex than what can be expressed in a single expression. It also is implemented in a different manner, since it is recompiled every time it is referenced. But it does allow us to take a look at the abstraction we can achieve by defining a general-purpose operation. We will then be able to extend this approach to defining more complex procedures.

A statement function is really a shorthand notation that allows you to define a complex operation only once in all its detail, and then refer to it simply

by name from then on. You might think of this as analogous to developing a *template*, or a form, similar to those for filing income taxes or applying for a security clearance. Once the form is there, all you have to do is plug in the appropriate values in the right places, and the job is complete. Analogously, you might define the operations involved in finding the hypotenuse of a right-angled triangle of short sides *a* and *b* just once, and then use the defined procedure to determine the hypotenuse for each of a variety of triangles.

The general form of a statement function is:

$$\text{fname}([\text{arguments}]) = \text{expression}(\text{arguments})$$

It is a non-executable reference statement provided to define a relationship among the arguments in its argument list. This relationship is defined by the expression on the right. The *arguments* are *dummy arguments*, that is, they are not actual variables (though they must follow the variable naming conventions) with values of their own. If the names of the dummy variables are unique, that is, not the same as any actual variables in the program unit (though they *could* be the same), they would not appear on a symbolic reference map of the variables that are allocated memory locations by the program. Instead, each dummy variable is a kind of place-holder in the expression, which gets its values when the statement function is actually referenced in the program. The expression on the right can be any legal expression that could appear on the right-hand side of an assignment statement.

## Statement Function for a Hypotenuse

For example, to define a statement function for finding the hypotenuse of a right-angled triangle of any pair of short sides *a* and *b*, we would write the *real* statement function:

$$\text{HYP}(A, B) = \text{SQRT}(A^{**2} + B^{**2})$$

The type of the statement function name itself determines the type of the value it will produce; thus, in this case we would calculate and return a real value. This statement function may be considered a general definition of the hypotenuse of any triangle with sides *A* and *B*. The *arguments* *A* and *B* are not actual variables. Instead, they *stand for* any two real values you want to plug into the expression. This would be done by an actual program reference to the statement function, such as:

```
PRINT*, HYP(3.0, 4.0)
Q = 7.5 + HYP(X, Y)
```

where the latter example assumes that values have already been defined for variables *X* and *Y* in the program. These two references to the HYP statement

function would then be compiled by implementing the formula defined under the name HYP, with the appropriate values substituted for A and B. Thus, the machine code generated for these two expressions would be identical to that which would have been created by the statements:

```
PRINT*, SQRT (3.0**2 + 4.0**2)
Q = 7.5 + SQRT (X**2 + Y**2)
```

This means that the statement function does not accomplish any savings in compiler time or memory space. It is, however, a convenience for the programmer and the program reader.

## Restrictions on Statement Functions

The expression in the statement function may involve constants, variables, array elements, the dummy arguments, and any system function (such as SQRT) or previously defined statement function, and must obey the rules for a properly formed expression. The statement function itself may be of type integer, real, double precision, complex, logical, or character, and the corresponding expression must use the proper operators for values of the specified type. Thus, we could define a concatenation operator for any three short character arguments:

```
CHARACTER CONCAT*10, C*3, D*3, E*4
CONCAT(C, D, E) = C//D//E
```

A statement function must appear at the beginning of the program, after any declaration statements (such as type statements, dimensions, COMMON, and the like) and before any executable statements; it may precede DATA statements. It may be referenced as many times as you like in the program, as long as the actual arguments provided to it agree in number and type with the dummy arguments in the definition. Of course, as we already mentioned, the severest restriction on a statement function is that it must be expressible as a single statement.

## Statement Functions for Temperature Conversions

Another example of a statement function would be to convert degrees Fahrenheit to degrees Centigrade, or vice versa. The formulas for these conversions should be familiar:

$$^{\circ}\text{F} = \frac{9}{5} ^{\circ}\text{C} + 32 \quad ; \quad ^{\circ}\text{C} = \frac{5}{9} (^{\circ}\text{F} - 32)$$

We could thus write the following statement functions:

```
FAHR(C) = 9.0/5.0*C + 32.0
CENT(F) = 5.0/9.0*(F-32.0)
```

Notice how nicely the statement functions parallel the formula representations of temperature conversion. These statement functions could then be referenced by statements in the program. We indicate several such references, followed by how they actually may be compiled in the program.

```
DO 40 CEN = 0.,100., 10.
    PRINT*, CEN, FAHR(CEN)
        {compiled as PRINT*, CEN, 9.0/5.0*CEN + 32.0 }
40  CONTINUE
    Q = CENT(212.0)
        {compiled as Q = 5.0/9.0 * (212.0 - 32.0)}
```

From these examples you can see how the actual arguments are plugged into the statement function for the dummy arguments and the entire line is compiled as if the full expression were present in the reference.

## “Helper” Statement Functions

A statement function may make use of any system functions, or any *previously defined* statement functions. Thus, we could have the following example, in which a “helper function” is defined first, and then used in the second statement function. The formula for monthly loan payments, for a principal P at a rate of interest of I% for N years is:

$$\text{monthly payment} = (P)(I/1200.)/(1-(1/(1+I/1200.)^{(Nx12)}))$$

Since  $I/1200.$  needs to be calculated twice, instead of writing it out in full two times, we can define a statement function for it:

$$\begin{aligned}\text{RATE}(I) &= I/1200. \\ \text{PAYMON}(P,I,N) &= P*\text{RATE}(I)/(1-(1/(1+\text{RATE}(I)))^{(N*12)}))\end{aligned}$$

This is a useful formula to have handy to estimate how much a car loan or house mortgage payments may be.

Another example in which you could use a “helper” statement function would be one where you wanted to determine the difference, in seconds, between two times expressed in military time (from 00:00:00 to 23:59:59). Your statement function would thus take as arguments six values, the time in hours, minutes, and seconds for the first instance, and the time in hours,

minutes, and seconds for the second event. Your statement function might well be given times such as:

```
PRINT*, TIME(9,45,56,14,33,21)
```

to determine the difference between 9:45:56 and 14:33:21.

Since the minutes for the later event might be less than those for the earlier, and the same might hold for the seconds, it would be simpler not to worry about borrowing minutes from the hours to subtract, especially since all of this must be done in a single arithmetic expression. Thus, an improvement in viewing the problem would be to write a helper function that would compute, for any specified military time, the total number of seconds elapsed from 00:00:00 to that time. Then all we would have to do is subtract the number of seconds elapsed to the first time from the number of seconds to the second event. The helper function would be written as follows:

```
HELP (A, B, C) = A*3600 + B*60 + C
```

to determine the number of seconds to time A:B:C. Then we can write our TIME statement function using this function. Note that the first event may not always be at the earlier time, as in the case of a reference to the statement function of the form:

```
AFTER = TIME (13,55,10,4,34,20)
```

Because we are just interested in the number of seconds between two events, no matter which one occurred first in time, we can just take the absolute value of the difference between the times in seconds:

```
REAL M1, M2
TIME(H1,M1,S1,H2,M2,S2) = ABS( HELP(H2,M2,S2) - HELP(H1,M1,S1) )
```

Since statement functions are generally compiled entirely "in-line" every time they are referenced, no savings in compile time or memory space may be accomplished by using one. The primary advantage of a statement function is to make life easier on the programmer, who then does not have to write out the full expression each time it occurs. A statement function may also be used to display prominently at the beginning of a program a very important formula that will be used in the program.

You should note that program variables may also appear in the statement function expression; for example,

```
EXAM(X) = A*X**2 - B
```

can use program variables A and B (defined before the reference).



## USE OF SYSTEM FUNCTIONS

The FORTRAN 77 compilers have a number of utility functions built in for your use: ones to take absolute values or square roots, or to compute various trigonometric functions such as the sine of an angle, and many others. A list of these available functions is given in Appendix B for your reference. Generally, the type of value returned by such functions is determined by the default type of the name, but there are a few exceptions. The function CHAR(N) returns the *character* that is in position N of the collating sequence, and there are several logical lexical functions (LLE, LGE, LLT, LGT) which return a *logical* value which is .TRUE. or .FALSE. depending on whether the two character strings provided to the function are in the indicated order.

Several functions in FORTRAN 77 are *generic*, which means that they can take a number of different types of arguments (usually integer, real, double precision, or complex) and return a value of the type of the argument. Thus, for example, the generic sine function SIN will take a real, double precision, or complex argument, and will return the appropriate real, double precision, or complex value of the sine.

A function reference may appear as part of a more complex expression. The value of the function is simply calculated at that point, and returned to the calculation, just as if it were a simple operation such as multiplication. Thus you can have:

```
PRINT*, ABS(B) + SQRT(SIN(C))
M = 100*TAN(A) + 1
```

and the function values will be returned and the calculation completed.

Such function notation is very convenient, since many scientific calculations involve functional references, and this allows you to write a representative FORTRAN statement incorporating a similar notation. We will see later how to write functions of our own.



## SUBROUTINES

A *subroutine* is a separate program unit that must be *called* by the main program (what we have been writing so far), but it need have no other connection with the main program. A subroutine can perform an independent task, or it can share values with the calling program. There are several advantages to using subprograms. They allow the programmer to break the program into separate tasks, or *modules*, easily, and they also allow the writing of general-purpose routines that may be used many times, operating on different values each time.

## Simple Subroutine Without Arguments

We will begin with a very simple subprogram, one which is written to perform a single task several times, but which requires no information from the calling program. We would like, at several points in our program, to print a header at the top of a new page. This header is to put the word 'CHAOS' at the top of the page, and then label all of the columns on the printer page, from 1 to 132. This operation involves several instructions and formats. If it were incorporated into the main program, it would either have to be written out completely each time it was needed, or else a complex system of GO TO's that would enter the code and then go back to the right place in the program would have to be worked out. Instead, we can write the appropriate instructions once, incorporate them into a subroutine, and then CALL the subroutine every time we need it.

A subroutine must begin with an entry line of the form:

SUBROUTINE name [(dummy arguments)]

and it must include at least one RETURN statement, and its own END statement for the compiler. If the RETURN is omitted, the compiler will interpret the END statement as acting as a RETURN. All variables (except dummy variables and those in COMMON) and statement numbers are *local* to the subroutine, which means that they have no relation to the variables and statement numbers in other program units (main program or subprograms). Thus, putting values into such local variables will not affect variables in other program units, even those with the same names, and statement numbers in the subroutine do not conflict with those in other program units. A subroutine is referenced by a statement:

CALL name [(actual arguments)]

in the calling program. If the subroutine has arguments, they must be provided in parentheses in the CALL statement.

Our simple header subroutine has no arguments, since its job is independent of any information which would have to be provided from the main program (except that of when to execute it, which is indicated by where the CALL appears in the main program). To create three-digit numbers for each column, we have used implied lists. The subroutine is:

```

SUBROUTINE THINK
PRINT 5
5  FORMAT('1', 63X, 'CHAOS')
      PRINT 15, (0, K = 1, 99), (1, L = 1, 33)
15  FORMAT(' ',132I1)
      PRINT 15, (0, K = 1, 9), ((N, K = 1, 10), N = 1, 9),
      & ((N, K = 1, 10), N = 0, 2), (3, L = 1, 3)

```

```

PRINT 15, (((N, N = 1, 9), 0), K = 1, 13), 1, 2
RETURN
END

```

The calling program would have several statements of the form:

```
CALL THINK
```

at different points to invoke the subroutine. When this statement is executed, control is passed into the THINK subroutine, and it begins execution; when the RETURN statement is encountered, control is transferred back to the calling program, to the statement *immediately following* the CALL statement. This may be accomplished by the use of a program counter that keeps track of the machine instructions being executed, such that the address of the CALL statement occurs at a particular value of the program counter (PC), and the RETURN then jumps back to address PC + 1 (the next instruction).

The output created as a result of each CALL to subroutine THINK would look like the following, at the top of a new page:

CHAOS		
0000000000 . . .	0011 . . .	1111
0000000011 . . .	9900 . . .	2333
12345678901 . . .	8901 . . .	9012

This subroutine allows easy determination of the proper placing of subsequent output on the page, since it clearly indicates each column number, and will aid in determining proper spacings.

## Adding Arguments to a Subroutine

Let us imagine changing the THINK subroutine so that it does not print 'CHAOS' on the top of every page, but some appropriate phrase describing what sort of output that page will contain. Then, we would like the calling program to be able to specify what would be printed at the top of each page, but the rest of the routine would be the same every time. We need to provide one *dummy argument* in the subroutine, which will receive the character string to be printed at the top of the page. We would thus modify the subroutine as follows:

```

SUBROUTINE THINK (TITLE)
CHARACTER*30 TITLE
PRINT 5, TITLE
5 FORMAT('1', 55X, A30)

```

- The rest of the subroutine would remain the same. CALLs to the subroutine

from the main program might be as follows:

```
CALL THINK ('POPULATION TRENDS')
...
CALL THINK ('PROJECTED ACID RAINFALL')
...
CALL THINK ('NUCLEAR REACTOR MEASUREMENTS')
```

So far, the *only* way for values to be shared between the calling program and the subprogram is through the argument list. Later, we will see another mechanism for sharing variables, that of using COMMON. Our first THINK example was independent of any information from the calling program, except for when it should execute, but the second THINK example took information from the calling program and used it. It is also possible for the subroutine to give a value to a variable in the calling program, or to modify a value passed to it by the calling routine. All dummy variables represent *shared values* with the calling program, and these values may be shared in either or both directions. We will speak, however, of values being *passed to* the subroutine, or being *returned by* the subroutine, depending on how they are used in the subroutine. There is no other means of distinguishing which values are passed in which direction in FORTRAN 77, so the programmer must be careful not to accidentally destroy values in the calling program that should be left alone.

The following simple subroutine indicates values being shared in all three ways between a calling program and a subroutine. Our subroutine will "accept" two values from the calling program, increment the second value by 1, and calculate and "return" the sum of these two values to the calling program. The subroutine will have three dummy arguments, A, B, and SUM, where A ("in") and B are values provided by the calling program, B is the value incremented in the subroutine (so it is both "in" and "out"), and SUM is the value returned by the subroutine (so it has the status "out").

```
SUBROUTINE ADDEM (A, B, SUM)
B = B + 1
SUM = A + B
RETURN
END
```

You can see from the use of the variables in the subroutine that A is merely referenced (it appears on the right-hand side of the = in an assignment statement), the dummy variable B is referenced *and* (re)defined, and the dummy variable SUM is just defined in the subroutine. Thus we might say that A and B are passed to the subroutine, that B is modified in the subroutine so that a new value is passed back, and SUM is defined in the subroutine and returned to the calling program.

CALLs to our example program might be:

```
CALL ADDEM (X, Y, Z)
CALL ADDEM (4.0, Q, R)
```

In the first example, the actual arguments passed are all variables, but in the second call, the first argument (corresponding to dummy argument A) is a constant. This is all right, since the subroutine does not modify this argument, it just uses it. You must be careful, however, not to allow the subroutine to alter such an argument, since Standard FORTRAN 77 provides no safety mechanism to protect against this, and thus the value of a "constant" might be changed (the effect is compiler-dependent). This could actually cause strange problems in your program, since often a compiler will store a common constant *once* and then refer to that location every time a reference to that constant occurs. In our case, if the constant 4.0 is stored in a certain location, a call to subroutine ADDEM changes it to a 5.0, and a subsequent use of the constant 4.0 goes to that location, you might have the unexpected result of

```
A = 4.0 + 4.0
```

storing a value of 10.0 into A!

Values may thus be shared, through the argument list, in three different ways. A value may be passed to the subroutine, and just used in its calculations, but not modified; such an actual argument may be a constant, a variable, or an expression. A variable may be passed to the subroutine and changed there, with a new value being passed back. And, third, a variable may be given a value (defined) in the subroutine, and passed back to the calling program. In the last two instances the actual argument provided in the CALL *must* be a single variable or array name. All dummy arguments in the subroutine are represented as variables, since they will take on values or identities depending on the calling argument list.

*Note:* In Fortran 90, arguments shared with subprograms will be able to be declared as "intent" IN (only to be used, but not modified), OUT (to be defined only), or INOUT (to be used and defined). See Appendix E for details on these declarations.

## Subroutines Which Provide Generality

A subroutine may be used simply to segment your program into manageable subtasks. In this case, some such tasks may be independent of what is going on in the rest of the program, such as THINK, and will need no arguments; others may need to operate on values contained in the main program or other subprograms, and thus arguments will have to be used. However, a subroutine

can do much more than just provide a mechanism for writing independent task modules. A subroutine can be written to provide a general-purpose service, on a variety of different sets of arguments in one program, or in many different programs.

Let us consider writing a subroutine to accept a one-dimensional real array filled with 100 values and calculate and return the average of the array values. The subroutine will need two arguments, one for the array and one for the average value to be returned. Since the first dummy argument is to represent an array, it must appear in a dimension statement in the subroutine. Since the subroutine is compiled completely independent of the calling program, this is the only way it will know that the argument is an array, and that it is legal to write it with subscripts. The logic of the subroutine itself is simple:

```
SUBROUTINE AVERAG (A, AV)
DIMENSION A(100)
SUM = 0.
DO 20 I = 1, 100
20    SUM = SUM + A(I)
AV = SUM/100
RETURN
END
```

Now, what is important to notice about this routine is that it can average *any* real, one-dimensional filled array of length 100. Thus we might have several calls to this subroutine:

```
REAL B(100), C(100), D(100)
... {arrays get filled with values}
CALL AVERAG (B, BAV)
CALL AVERAG (C, CAV)
CALL AVERAG (D, DAV)
```

and it could even prove useful in other programs we write. In such a case, the code would merely need to be copied into the file containing the new program, since it has already been checked and seen to work. Notice that in passing an array as an argument, only the name of the array is used. In this example, we see that our subroutine has given us a level of generality that we could have never achieved with one set of code if we had only one (main) program unit, no matter how fancy we made the entries and exits—such code could only have worked on *one* array, instead of many different ones, as our subroutine can.

We might like to expand our level of generality even further—why restrict ourselves just to arrays of length 100? Why not a routine which would average any real one-dimensional array of *any length*? We would have to pass one more argument to the subroutine, and use it in all of the places where the constant

100 appeared in our earlier version. Can we do this? The only area that seems questionable is that of the array dimension.

## Variable Dimensioning

We saw that we had to put our dummy array argument name in a dimensioning statement so that the subroutine would recognize it as an array, and would consider subscripted references to it legal. However, the dummy array itself occupies no actual locations; it merely makes it possible to associate references to it with an actual array in the calling program. Thus the actual *size* to which it is dimensioned is irrelevant (for one-dimensional arrays), since the compiler does not have to set aside any actual space for it. We could have written DIMENSION A(100) or DIMENSION A(1) or DIMENSION A(500) or DIMENSION A(\*) or even DIMENSION A(N). In this last case, N would have to be another dummy argument to the subroutine, representing the actual length of the array passed. Such a reference is called *variable dimensioning*. The only place we could get away with this is in a subprogram, where the array and its dimension are both dummy arguments to the subprogram. You can see that you could not use a variable to dimension an actual array, since then the compiler would not know how much space to set aside for it. In the one-dimensional array case, we can use either A(N) or A(\*)�.

Thus, the subroutine could be rewritten:

```
SUBROUTINE AVERAG (A, N, AV)
DIMENSION A(N)           {or A(*)}
SUM = 0.
DO 20 I = 1, N
20   SUM = SUM + A(I)
AV = SUM/N
RETURN
END
```

Now we can use this general-purpose subroutine to average *any* real one-dimensional array of *any length*; this makes it even more useful than before. Calls to this subroutine might be:

```
DIMENSION A(200), B(400), C(350)
... {arrays are filled}
CALL AVERAG (A, 200, AAV)
CALL AVERAG (B, 400, BAV)
CALL AVERAG (C, 350, CAV)
```

Similarly, we can write general-purpose subroutines to handle two-dimensional arrays of any size. In this case, we will also use variable dimensioning.

We *could* use an asterisk (\*) in the column dimension, but we *must* specify the row dimension, using a dummy variable, since the computer must make use of that information in calculating the position of an array element. In a two-dimensional array A, location A(I, J) is calculated to be in the following position in the array:

$$(J - 1) * (\# \text{ of rows}) + I$$

since two-dimensional arrays are stored in column-by-column order in FORTRAN 77. Thus the machine must know how many rows this particular configuration has in order to calculate its array positions.

We could, for example, write a subroutine to fill any two-dimensional integer array (of M rows and N columns) with the integers from 1 to MxN, in sequential order across the rows.

```
SUBROUTINE FILLER (MACK, M, N)
DIMENSION MACK(M, N)
K = 1
DO 30 I = 1, M
    DO 30 J = 1, N
        MACK(I,J) = K
        K = K + 1
30 CONTINUE
RETURN
END
```

Notice that this subroutine does not have to have an integer-type name, even though it is designed to fill integer arrays. The default type of a subroutine name has absolutely no effect on the types of values it may process. We shall see later on that the case is different with functions.

This subroutine could then be used to fill any two-dimensional integer array. Sample calls to the subroutine might be:

```
INTEGER CAT(20, 30), DOG(10, 25), INK(24, 40)
CALL FILLER (CAT, 20, 30)
CALL FILLER (DOG, 10, 25)
CALL FILLER (INK, 24, 40)
```

Notice that a subroutine may be called from a main program, or from another subprogram; it just may not call itself in Standard FORTRAN. A procedure that calls itself is said to be *recursive*; the capability to define such procedures exists now in several FORTRAN compilers, but it is not standard. Check to see if your compiler allows recursion. Recursion will be part of the Fortran 90 Standard.



## COMMON (GLOBAL VARIABLES)

There is another means of sharing variables among program units, that of declaring them to be stored in a special "common" area of memory. Then any program unit containing a COMMON statement has access to this area of memory and the variables stored in it. References to variables stored in a common area are more direct, and thus faster, than references made through the use of dummy arguments in a call list.

### Blank COMMON

A simple COMMON statement, for "blank" COMMON, is of the form:

COMMON list\_of\_variables

and it designates that the variables on the list, *in the order they are mentioned*, are stored in a particular area of memory called blank COMMON. This area, because it is usually designated to be in the area of memory *below* the program proper, where the loader resided when loading in the program, can be used to extend the normal memory capacity of the program (by using locations that would otherwise be ignored at execution time). However, because these locations do not exist at compile time, they may not be initialized using DATA statements.

A COMMON statement may also be used to declare array dimensions, just as you can do in a type statement. An example COMMON declaration would be:

COMMON A, B(50), C, LATE(4, 5)

Then if a similar COMMON statement appeared in another program unit, it would have access to the same locations. The best and safest method is for the COMMON statements to be identical in the two (or more) program units, using the same variable names, but this is not necessary. Corresponding COMMON statements might be:

COMMON X, Y(50), Z, MIKE(4, 5)	{in subroutine ONE}
COMMON ALL(72)	{in subroutine TWO}

Since these variables name the locations in blank COMMON, *in order*, they refer to the same locations as the first COMMON statement. Thus variable X in the second program unit is the same as variable A in the first, and so on. A "memory map" of blank COMMON as defined by these COMMON statements would be:

blank COMMON		
{in Main program}	{in subroutine ONE}	{in TWO}
A	X	ALL(1)
B(1)	Y(1)	ALL(2)
B(2)	Y(2)	.
.	.	.
.	.	.
B(50)	Y(50)	.
C	Z	.
LATE(1,1)	MIKE(1,1)	.
LATE(2,1)	MIKE(2,1)	.
.	.	.
.	.	.
LATE(4,5)	MIKE(4,5)	ALL(72)

Though this is potentially confusing, and so not recommended, it is the way COMMON declarations are interpreted. It does allow program units written by different programmers to use different names, as long as the COMMON blocks are structured the same way. However, we strongly recommend using identical COMMON statements, if you are using COMMON.

A COMMON block may be shared by as many subprograms as you wish. If a program unit only needs the first few elements in the blank COMMON list, the block only need mention the elements on the list up to those to be referenced. However, even if the program unit only needs the last element of the COMMON block, the entire contents of the block must be listed, so that its proper location is found. In general, variable types may be mixed in COMMON, with one exception—if there are any character variables in a COMMON block in FORTRAN 77, there may be no variables of other types in that same block. (This restriction was not in force in earlier versions of FORTRAN, however, before the CHARACTER data type was introduced.) This is because the storage conventions for character variables are not standardized from one machine to the next, and so the space they will take up cannot be predicted for purposes of association of memory areas.

## Labelled COMMON

Blank COMMON is not the only special area available to store shared variables. You may create any number of areas of *labelled* COMMON by including an area name (following the usual naming conventions) between slashes in your

COMMON statement. Thus, if the first set of variables is unlabelled, they will go into blank COMMON, and each labelled set of variables will go into an area identified by that label. Thus, in the following:

```
COMMON A, B, D(5) /AREA1/ C, KATE(6), /CHESS/ BOARD(8,8)
```

blank COMMON will contain the real variables A and B and the real array D, the labelled COMMON area called AREA1 will contain C and the array KATE, and the labelled area called CHESS will contain the two-dimensional array BOARD. A comma may optionally be used to separate COMMON areas being defined.

There are several advantages to using labelled COMMON. If any character variables are to be put into COMMON, no other types of variables may go into the same area; this makes it advantageous to be able to declare many different areas of COMMON and not just use blank COMMON. Furthermore, if a certain subprogram only needs access to a few variables in COMMON areas, and not to all of them, then that subprogram only has to include the reference to the particular area(s) that are relevant.

Labelled COMMON areas go into different spaces in the program storage area, not into the area which held the loader. Thus, they may be initialized using DATA statements, but only (in the FORTRAN Standard) by using a BLOCK DATA subprogram.

## BLOCK DATA Subprograms

Variables in labelled COMMON areas *can* be initialized using DATA statements, but the Standard says this must be done in a special subprogram unit, the BLOCK DATA subprogram (though some compilers do not impose this restriction). This subprogram must be a separate program unit, and begin with the entry line:

```
BLOCK DATA [name]
```

where the name is optional. There may be more than one BLOCK DATA subprogram in a program, and then the name allows them to be distinguished from one another. The name, if used, must not be the same as that of any other subprogram, or the name of the program (if any), or the name of any COMMON block, or of any local variable in the subprogram itself.

The BLOCK DATA subprogram then contains all relevant type, dimension, and COMMON statements for the variables to be initialized, DATA statement(s) for the initialization, and an END statement (no RETURN). Thus, a simple example would be:

```
BLOCK DATA ONE
```

```

CHARACTER*5 C(8), D, X(10)*7
INTEGER CAT(100)
COMMON /CC/ CAT, DOG /CHARS/ C, D, X
DATA CAT/100*2/, C/'A','B','C','D','E','F','G','H'/
END

```

## Encoding Problem Using COMMON

An entertaining problem that can be implemented with COMMON is the following. Suppose that you want to send an encoded message home from college. You want your father, who is a math and puzzle buff, to be able to decipher it, since he is also the "soft touch" in the family. You want your mother just to think it is just an arithmetic exercise. The message you want to send is:

```

SEND
MORE
MONEY

```

The coding scheme is as follows. You want to code each letter of the message into a different unique digit, 0–9. Further, there should be no leading zeros in the message, so S and M cannot code into 0, and, after the coding is finished, the numbers should work out to an actual addition problem. That is, the four-digit numbers that SEND and MORE have been coded into should, when added together, equal a result which is the same as the five-digit number into which MONEY was coded.

You may want to try out this problem on your own before looking at the computer solution we are about to give. Such problems are fun to work out if you like number puzzles. If so, put down the book at this point and try to find the solution using your grey matter.

Did you get it? Or did you give up? In either case, here is the computer's solution to the problem. We could have had the program use more nested loops to solve the problem from scratch, but we did some initial analysis on the problem, and determined what some of the letter values had to be before we started writing the program. The "M" in MONEY must come from a carry arising from adding the S and M in the fourth column. Since there are only two digits in that column, M can only be a 1, not a 2. If M is 1, then S + M (plus perhaps a carry digit from the third column) must be 10, or 11, or 12. The sum in the fourth column cannot be 11, since that would make the letter O a 1, which it cannot be, since we know that M is 1. Even S + M + carry digit can't make 12, so the sum in the fourth column must be 10, and thus O codes into a 0 (zero). If there is no carry digit, then S will be a 9; otherwise it will be an 8. Can there be a carry digit from column three? It involves adding E + O (which is zero), plus (perhaps) a carry digit, to get N. This means there must be a carry digit from column two, and E + 1 (carry) + 0 (letter O) must be N

(not  $N + 10$ ). Thus, there is *no* carry digit from this column, meaning that  $S$  must be 9. We have also found the relationship that  $E + 1 = N$ .

At this point, we decided to give up our human mental gymnastics (which saved us three nested loops), and have the computer do the rest of the work. Given the starting points we had already determined, we then wrote a program to try out all of the possible other codings of letters (R, D, Y, N, and E) into digits (the remaining digits are 2–8). In this way, we can also determine if there is more than one solution to the puzzle (which we might not be sure of if we just came up with one ourselves).

We thus set up nested loops to try out all possible mappings of the letters onto digits, with the constraints that no two letters can map onto the same digit, and that  $E + 1 = N$ . We used a subroutine, CALC, to take the codings and see if they make the arithmetic problem of addition work. This was done to modularize the code, rather than to have a general-purpose utility subroutine. The resulting program is the following.

```

PROGRAM MESSAGE
COMMON S, E, N, D, M, O, R, Y
INTEGER S, E, N, D, M, O, R, Y
PRINT 1
1 FORMAT(////'0', 40X, 'SOLUTIONS')
M = 1
O = 0
S = 9
DO 99 R = 2, 8
  DO 98 D = 2, 8
    IF (D.NE.R) THEN
      DO 97 Y = 2, 8
        IF (.NOT.(Y.EQ.R .OR. Y.EQ.D)) THEN
          DO 96 N = 3, 8
            IF (.NOT.(N.EQ.R .OR. N.EQ.D .OR. N.EQ.Y)) THEN
              E = N - 1
              IF (E.NE.R .AND. E.NE.D .AND. E.NE.Y) THEN
                CALL CALC
              ENDIF
            ENDIF
          CONTINUE
        ENDIF
      CONTINUE
    ENDIF
  CONTINUE
ENDIF
97 CONTINUE
98 CONTINUE
99 CONTINUE
STOP
END

```

```

SUBROUTINE CALC
COMMON S, E, N, D, M, O, R, Y
INTEGER S, E, N, D, M, O, R, Y, SEND, MORE, MONEY, MADD
SEND = S*1000 + E*100 + N*10 + D
MORE = 1000 + R*10 + E
MONEY = 10000 + N*100 + E*10 + Y
MADD = SEND + MORE
IF (MADD .EQ. MONEY) PRINT 5, SEND, MORE, MONEY
5 FORMAT('0',43X, I4/'0',42X, '+',I4/43X,'____'/'0',42X,I5)
RETURN
END

```

It was much simpler to concentrate on the details of the mapping of letters into 4- or 5-digit numbers, and check out the addition, as a separate program unit. The letters which have been coded are put in blank COMMON, since it is simpler and easier to share them with the subroutine that way than it would be to put eight arguments in the calling sequence. The program then prints out *all* solutions it finds.

The resulting solution (there was only one) printed out is:

SOLUTIONS	
9567	
+ 1085	
<hr/>	
10652	

Since the variables in COMMON were all integer, not character, there was no problem of mixing character data types in a COMMON area with non-character data types. The main program does the work of setting up the mappings of letters onto digits, and the subroutine CALC does the job of evaluating the mapping. The use of a subroutine represents a good division of labor—it is easier to write and easier for someone to read and follow at a later date.

## Advantages and Disadvantages of COMMON

Usually, more generality can be achieved in subprograms by using an argument calling list rather than COMMON, and this makes each subprogram more independent and less subject to accidental changes by other program units. Considerations of what is called *information hiding* are becoming more prevalent these days, particularly in sensitive problem application areas such as defense, suggesting that it is advisable to make the internal mechanism of a subprogram opaque to the user. The program using the subroutine can know *what* it does, but not *how* it does it.

However, there may be certain cases where the use of COMMON is clearly warranted, since references to variables in COMMON are more direct and efficient than to those in argument lists. Imagine, for instance, an elaborate chess-playing program which utilizes many subprograms for different phases of the game, such as OPEN, CHECK, MIDBRD, and so on. It would be sensible, since all of these routines deal with the *same* board, and changes made to the board by one subprogram should be transmitted to the other subprograms, to put the chess board itself into COMMON. For example, each subprogram could contain the COMMON statement, and its related type statement:

```
INTEGER BOARD
COMMON BOARD(8,8)
```

*Note:* In Fortran 90, a programmer may use the *module* construct, which will make the use of COMMON outmoded. A module may be used to declare types and dimensions of variables, may contain module subprograms, and may be used to define new data types (see Chapter 13). The contents of a module may be accessed by any program unit, through an appropriate USE statement. A module definition has the form:

```
MODULE name
  [module specification statements (type, dimension)]
  [module subprogram definitions]
END MODULE [name]
```

and its contents may be included in any program unit by a

```
USE name
```

statement. The module name is global, and thus may not be the same as any other global variable or program unit name. If any variable is declared PRIVATE in the module declaration area, it is not available outside of the module. This allows for *data encapsulation*—the definition of data is accomplished in one place—and *data hiding*—the ability to keep information which is not necessary for operation secure from programs that use the module. An example of a module definition would be:

```
MODULE CLEAR
  REAL, PRIVATE :: X, Y, Z
  INTEGER MADDER, MUSIC (100)
  CONTAINS
    FUNCTION ASK (X, Y, Z)
      ...
      ASK = ...
    END FUNCTION ASK
  END MODULE
```

The information in module CLEAR (except that which is declared PRIVATE) is available to any program unit through a USE CLEAR statement. Notice that a module may define many variables and arrays, and that these may all be made available to any program unit making use of a USE statement, without having to list them.

## FUNCTIONS

A *function* is a subprogram whose primary purpose is to return one single value. It is similar in this way to a statement function, but it does not have to be a “one-liner.” A function may be as complex as you like, containing loops, array references, Block IF tests, and so on. Thus, you could write a function for any calculation whose main purpose is to calculate some single result.

Arguments are shared between a calling program and a function just as in a subroutine CALL, except that the function reference is simply made by naming the function and giving its list of actual arguments in parentheses following the name; e.g.,

```
SAVE = FUN(A, 5, R + 3.)
```

is a reference to function FUN, which begins:

```
FUNCTION FUN (X, N, Y)
```

If a type is not declared for a function, it takes on a type according to the implicit typing rules. This fact is important, since the type of the function name determines the type of the value it will return. If you *do* want to declare a type for the function, it may be done in one of two ways:

```
CHARACTER FUNCTION CAT(N, M)
```

{or}

```
FUNCTION CAT (N,M)
```

```
CHARACTER CAT
```

If the type of a function is not implicit, then it must also be declared in *every* program unit (main program or subprogram) that references it, or the values you return will be garbage.

Each function should contain one or more RETURN statements, and it must end with its own END statement. In addition, at some point in the program logic of the function it must store a value into the function name: e.g., FUN = ... The value stored into the function name is the value returned when RETURN is executed.

## Availability of Functions to the Rest of the Program

A function is available to any program unit in your workfile (unlike a statement function, which may only be used in the program unit in which it is defined). A function is only compiled once, but it may be entered as many times as you like. Its use is like that of a system function such as SQRT or MOD or SIN; you may make a function reference part of a more complicated calculation, and (at the point where it appears in the expression) a transfer of control is made to the function, its value is calculated, and the value is returned to the exact place in the calculation from which the transfer was made.

## Comparison of Subroutines and Functions

Compare the use of a subroutine and a function to do the same job—to find the greatest common divisor of any two integer arguments. We will assume that this result is then to appear in a calculation.

```
CALL GCD (M, N, MN)
MARY = 3*IGCD(M,N)-5
MARY = 3*MN-5
```

We see that the function reference is more direct, and that it does not require the use of an additional variable location in which to store the result. Thus some of the examples we used for subroutines, such as that of finding the average value of a real one-dimensional array of any size, are really more appropriately performed as functions. We would simply modify our AVERAG subroutine by changing it to:

```
FUNCTION AVERAG (A, N)
REAL A(N)
SUM = 0.
DO 20 I = 1, N
20    SUM = SUM + A(I)
AVERAG = SUM/N
RETURN
END
```

To construct another function from the beginning, let us write a function to find and return the biggest value in any two-dimensional integer array of any size (think of variable dimensioning for this, just as we used in subroutines):

```
FUNCTION LARGE (KAT, M, N)
DIMENSION KAT (M, N)
```

```

LARGE = KAT(1, 1)
DO 20 I = 1, M
    DO 20 J = 1, N
        IF (KAT(I, J) .GT. LARGE) LARGE = KAT(I, J)
20 CONTINUE
RETURN
END

```

This function could then be referenced with any of the indicated filled arrays as arguments:

```

INTEGER MAD(5, 6), COOL (10, 20), CAT (100, 30)
...
PRINT*, LARGE (MAD, 5, 6)
METWO = LARGE (COOL, 10, 20) + LARGE (CAT, 100, 30)

```

Entries to, and returns from, functions work just as we have described for subroutines, except that the RETURN brings you back to the point in the machine language expansion of the statement in which the function reference occurred, ready to continue the calculation or other operation involved.

## Function to Test for Identical Matrices

You could, for example, write a function to accept two integer two-dimensional arrays of the same size, whose dimensions would be specified, and determine if they are identical or not; return a 'YES' if they are identical, a 'NO' if not.

```

CHARACTER*3 FUNCTION COMPAR (ONE, TWO, M, N)
INTEGER ONE(M, N), TWO(M, N)
DO 50 I = 1, M
    DO 50 J = 1, N
        IF (ONE (I, J) .NE. TWO (I, J)) THEN
            COMPAR = 'NO'
            RETURN
        ENDIF
50 CONTINUE
COMPAR = 'YES'
RETURN
END

```

Functions may make use of COMMON (blank or labelled), but more rarely than subroutines do. This is true because a function is generally designed to perform a *utility* operation, one which will be usable in a wide variety of programs and installations. The use of COMMON requires careful coordination between (or among) the program units. We would usually prefer to write a function which any program could use, without having to deal with any

special considerations. Thus, the argument list should, in most cases, create all the channels of communication between the user program segment and the function, and it is highly advisable that the function not change any of its arguments (though FORTRAN 77 provides no protection against this).

The set of system functions available in FORTRAN (in Appendix B) give a good range of examples in which a function is particularly useful. Although these functions already exist and have been well checked out, they provide you with a ready-made list of exercises to try—write your own versions for the system functions, and then compare your results to theirs. You can also write functions to determine any single-valued result for an array of values (of various dimensions), such as sum, average, standard deviation, minimum, maximum, or mode (most frequently occurring value), as well as providing counts of how many values in the array are equal to a target value or lie in a certain specified range of values.

A function reference may appear anywhere in a FORTRAN expression, or anywhere a FORTRAN expression may occur. Thus function references may occur in PRINT statements, arithmetic expressions in assignment statements, or as arguments to another subprogram. In the latter case, any functions are evaluated, and those values are passed to the subprogram.

```
PRINT*, FUN(X, Y, Z)
REST = 33.0*FUN(X,Y,A) + FUNNY(I,J)**2/24.0
CALL MAKEUP (A, FUN(3.0,4.0,5.0), 7, FUNNY (M, 5) )
```

Notice that in the last case, it is a function's *value*, not the function itself, that is passed as an argument to subroutine MAKEUP. Before the CALL to MAKEUP is executed, the function FUN must be evaluated for arguments 3.0, 4.0, and 5.0, and the function FUNNY must be evaluated for arguments M and 5. These values are then passed as arguments to MAKEUP.

A subprogram *name* may also be passed as an argument to another subprogram, as long as proper procedures are followed.



## SUBPROGRAMS AS ARGUMENTS (EXTERNAL, INTRINSIC)

We have already seen the great level of generality that subprograms can provide us. They can be written to perform very general jobs, such as average any array of a particular type of any size. They save much repetitive labor by providing general solutions to problems. We can extend their flexibility one step further. They can be written to perform operations on other subprograms, where the subprogram to be worked on can be varied, just as which array is passed can be varied.

For example, if we wanted to write general-purpose routines to perform numeric integration or differentiation of functions, or to determine the maximum value of a function in a specified range, or the like, we would not want to have to write a separate program for each function we wanted to work on. It would be best to be able to write the routine to perform, say, integration, and then allow it to work on a whole range of different functions. This can be done if we can pass the function *name* as an argument to the subprogram; then it can work on whatever function is passed to it.

Let us assume we want to write a function which will find the maximum value over a specified range of any real function of one argument passed to it. It will accept the name of the function, the endpoints of the range to be examined, and an integer argument to specify the number of points in the range to be examined. The function itself would look like the following:

```
***** FIND THE MAXIMUM OF A FUNCTION *****
FUNCTION FINDMX ( FUN, A, B, N)
***** FUN IS THE DUMMY NAME FOR THE FUNCTION PASSED *****
*** A AND B ARE THE ENDPOINTS OF THE RANGE TO BE EXAMINED ***
*** N IS THE NUMBER OF POINTS IN THE RANGE TO BE EXAMINED ***
      STEP = (B - A)/N
      FMAX = FUN(A)
      X = A
      DO 20 I = 1, N
          X = X + STEP
          VAL = FUN(X)
          IF (VAL .GT. FMAX) FMAX = VAL
20    CONTINUE
      FINDMX = FMAX
      RETURN
      END
```

This function can then be used to find the maximum value of *any* real function of one argument, as long as the proper conventions are followed when the function is referenced in a program unit. Since no variable names are "reserved" in FORTRAN, when a function name is to be passed to this function as an argument, the passing program unit must be put "on notice" that this parameter has a special use—that of providing the name of some subprogram as an argument to another subprogram. This must be done in FORTRAN by declaring in the calling program unit those function names that will be passed as arguments as belonging to a special category—"external" for user-defined subprograms, and "intrinsic" for system subprograms (mostly functions). Note that, on some systems, EXTERNAL may be used to cover both types.

Thus in the program we might use to call our FINDMX function, if we passed it a mixture of user-defined functions (such as FUNNY) and system functions,

they would have to be appropriately declared at the beginning of the calling program unit with EXTERNAL or INTRINSIC statements (to appear with the rest of the type, dimension, and COMMON declaration statements):

```
EXTERNAL FUNNY
INTRINSIC SIN, COS, EXP
A = FINDMX (FUNNY, 45., 99., 100)
B = FINDMX (SIN, 0.5, 1.02, 50)
C = FINDMX (COS, -0.4, .79, 75)
D = FINDMX (EXP, 1.0, 4.8, 60)
```

This ability to pass subprograms as arguments is especially useful in doing various numerical analysis problems, some of which we will discuss in a later chapter. Note that some compilers may allow *all* functions used as arguments to be declared as EXTERNAL, but this is not part of the Standard.

## ITERATIVE FUNCTIONS

Some functions (including some of the system utility functions) represent only approximations, rather than precise values. This is the case when the best expression we have for the function is either an approximating formula, or else is an infinite series. An example of the former would be the approximating formulas for square root and cube root, which are as follows:

$$\begin{array}{ll} \text{Square Root of A} & \\ X' = (X + A/X)/2.0 & \end{array}$$

$$\begin{array}{ll} \text{Cube Root of A} & \\ X' = (2X + A/X^2)/3.0 & \end{array}$$

Each new approximation,  $X'$ , comes from the previous approximation,  $X$ , by an appropriate manipulation. This means that we need a *first approximation*, or first guess, to start things off. The value whose square root or cube root we are looking for,  $A$ , could be the first guess, or  $A/2.0$ . The iterative technique homes in rapidly on the correct value, so that the first guess is not crucial to the operation. Notice the great similarity between the approximations for square root and cube root; once you have written one, the other would be very easy to write.

The question still remains, how long does one continue taking approximations? This is generally answered by, "As long as they continue to make a significant difference." But how do we assess such significance? If the new approximation,  $X'$ , is not *very* different from the old  $X$ , then there is not much point in continuing the approximation. One way to test this is to look at the ratio of the two values; if the ratio is very close to 1, then they are nearly equal, and there is no point in going any further. This can be done by looking at the

absolute value (why absolute value?) of the difference between the ratio and 1; if it is less than some small tolerance level, epsilon (e), that you set, then the approximation should be terminated:

$$\left| \frac{X}{X'} - 1.0 \right| < e$$

Given all of this information, we can now write an iterative function of our own to replace the system function SQRT. There are no reserved words in FORTRAN; thus, if we want to call our function SQRT, we may do so. However, if we do, it will *replace* (for our program only) the system function. If we want to retain the system function, perhaps so we can compare its results to our own, we must give ours a different name. We will do this. We will write our function to take the square root of any specified *real* argument. Thus, when using our function, we must be sure always to provide it with real values to work on.

```

FUNCTION SQROOT (A)
DATA EPSI /0.0001/
***** FIRST APPROXIMATION X IS A/2 *****
X = A/2.0
***** ITERATE TO NEXT APPROXIMATION, X' *****
30 CONTINUE
    XP = (X + A/X)/2.0
***** IS THE APPROXIMATION GOOD ENOUGH? *****
    DIFF = ABS(X/XP-1.0)
    X = XP
    IF (DIFF.GE.EPSI) GO TO 30
***** TRY AGAIN; NEW APPROX. BECOMES OLD APPROX. *****
    SQROOT = XP
    RETURN
END

```

Other iterative approximations are derived from infinite series expressions for the value sought. Some of the more familiar are the following, for pi and various functions:

$$\pi^2 = 6 + 6/2^2 + 6/3^2 + 6/4^2 + \dots$$

$$\pi = 4 (1 - 1/3 + 1/5 - 1/7 + 1/9 \dots) \quad [\text{Leibniz}]$$

$$\pi = 4 \times 2/3 \times 4/3 \times 4/5 \times 6/5 \times 6/7 \times \dots \quad [\text{John Wallis}]$$

$$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$$

$$\cos(x) = 1 - x^2/2! + x^4/4! - x^6/6! + \dots$$

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + \dots$$

In most of these series (except Wallis' multiplicative one), each new approximation comes from the old one by adding (or subtracting) a new term. The old and new approximations are compared in the same way as we did for square root, and, if the tolerance level is met, the iteration is stopped. In Wallis' formula, each new approximation comes from the old one by multiplying in a new term, and so a better termination test would be simply if the absolute difference of the two approximations was less than epsilon. Notice that some of the series involve alternating sums; you would thus have to figure out how to add, then subtract, terms on alternate passes through the routine. We suggest the use of a "flag" or "switch" value. If you set the value of a switch variable K to 1 when you are supposed to add a term, and to 2 if you are supposed to subtract a term, then you can proceed as follows, assuming the first term is to be added:

```

K = 1
SUM = 0.
20 CONTINUE
TERM = { . . . }
IF (K.EQ.1) THEN
    SUM = SUM + TERM
    K = 2
ELSE
    SUM = SUM-TERM
    K = 1
ENDIF
etc.
```



## USING SUBPROGRAMS EFFECTIVELY

As you continue programming, it is very likely that the programs you write will get to be of larger and larger scope. The only sensible way to approach a large problem is to break it up into smaller, manageable problems. The use of subprograms in FORTRAN allows you to do this elegantly. If the smaller problem involves returning some single value, such as the sine of an angle, or the average value of an array, or the highest jump recorded, then use a function. A function can be called more efficiently, its value returned more directly, and it is more flexible in its usage (that is, a function reference can appear in a more complex expression). For other subproblems, use a subroutine. Keep your subprograms as generally useful as possible. Only use COMMON for situations where it is clear that the variables in COMMON are the only ones you will ever want to work on, as, for example, putting a chess board in COMMON for many subroutines to examine that are part of a large chess-playing program (some of these are very successful!).

Try to minimize the number of arguments needed by your subprogram. Do not make the user supply a long list of arguments if it is not necessary; it is easy for the user to get mixed up in order, type, and number with a great many arguments. For example, if you are writing a function to convert a number from decimal to Roman numerals (or vice versa), the function should really only need *one* argument, the value to be converted. Any other reference arrays and the like should be provided within your subprogram. Subprograms should be "user-friendly."

A rather simple example of a larger-scale problem that is readily divided into parts is that of analysing a set of data read in from scientific measurements that are made. We would like our program to read in the measurements (we do not know ahead of time how many there will be; they are terminated by a "flag," negative value), average them, calculate the standard deviation, find the largest and smallest values, sort them into descending order, find the median, and then print out a histogram of measurements falling into specified subranges, each 10% of the overall range of the values. This project is readily broken up into parts. We will have the main program read in the data and determine how many values there are. We will then use subroutines or functions to do the other jobs. We will write one of the subprograms here, but the rest are ones you could readily fill in yourself or we have written earlier. The overall structure of the program would be something like this:

```

PROGRAM MEASUR
REAL VALUE(1000)
***** READ IN MEASUREMENTS UNTIL FLAG IS FOUND *****
N = 0
5  READ*, X
   IF (X.GE.0) THEN
      N = N + 1
      VALUE(N) = X
      GO TO 5
   ENDIF
   AV = AVERAG(VALUE, N)
   SD = STDDEV(VALUE, N, AV)
   CALL SORT(VALUE, N)
***** VALUES ARE NOW SORTED INTO DESCENDING ORDER *****
   BIG = VALUE(1)
   SMALL = VALUE(N)
   RANGE = BIG-SMALL
   IF ( MOD(N,2).EQ.0 ) THEN
      AMED = ( VALUE(N/2) + VALUE(N/2 + 1) )/2
   ELSE

```

```

        AMED = VALUE (N/2 + 1)
ENDIF
PRINT 77, AV, SD, BIG, SMALL, AMED
77 FORMAT (... )
CALL HISTO (VALUE, N, RANGE)
STOP
END

SUBROUTINE HISTO (Z, N, RANGE)
REAL Z(N)
CHARACTER ARR(100)
DATA ARR/100*'*/'
PRINT 5
5 FORMAT('1', 50X, 'HISTOGRAM'//'0', 4X, 'VALUE RANGE')
DIFF = RANGE/10.0
X = Z(1)
K = 1
DO 100 I = 1, 10
    Y = X-DIFF
    NUM = 0
60 IF (Z(K).GE.Y) THEN
    NUM = NUM + 1
    K = K + 1
    GO TO 60
ENDIF
K = K - 1
PRINT 75, Y, X, (ARR(L), L = 1, NUM)
75 FORMAT('0',F6.1,' - ',F6.1,2X,100A1)
X = Y
100 CONTINUE
RETURN
END

```

We have not supplied the functions to average or calculate the standard deviation, or the sort subroutine, but you could readily implement these yourself by now. The overall form of the big problem is already sketched out, and the one subroutine that was a somewhat new problem has been written; filling in the rest of the details is merely a matter of careful "bookwork." The advantage of dividing up a problem into subproblems is that we can come to a command of the problem rather quickly, and the elaboration that may remain can often be accomplished by utilizing subprograms (such as for averaging, standard deviation, or sorting) that we have already written for earlier jobs.

## Using SAVE

The FORTRAN Standard says that local variables in a subprogram, and values in labelled COMMON blocks that do not appear in the main program, are *undefined* upon leaving the subprogram. Though few processors in fact implement this restriction, FORTRAN 77 has provided a method of *saving* such values just in case. Since you may want certain local subprogram values to be retained from one call to the next, you may identify these variables in a SAVE statement at the beginning of the subprogram:

SAVE list

where the *list* may contain local variable and array names, and/or labelled COMMON block names, within slashes. If you use the SAVE statement in a subprogram without any list, it will save *all* “threatened” variables in the subprogram.

*Note:* Many FORTRAN 77 systems do not implement the feature of wiping out the values of local subprogram variables on the RETURN, and thus SAVE is often not needed.



## FORTRAN 90 FEATURES

Fortran 90 will allow the declaration of subprogram arguments as IN, OUT, or INOUT, thus monitoring whether they can be changed in the subprogram. These are declared using an INTENT statement in the subprogram; for example,

```
SUBROUTINE FIX (A, B, C)
REAL, INTENT (IN) :: A
REAL, INTENT (INOUT) :: B
REAL, INTENT (OUT) :: C
    B = B + 1.0
    C = A + B
RETURN
END
```

A *module* may be used in Fortran 90 to define a set of variables and arrays, which then may be accessed by any program unit through a USE statement. This allows the definition of a set of variables and arrays only once that then may be used throughout the program. See the discussion in the chapter and in Appendix E.

Fortran 90 allows the writing of *recursive* subprograms; see problem 25 at the end of the chapter.



## SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

You can modularize your programs, utilizing top-down design, by breaking a large problem into manageable subproblems and solving each of these subproblems separately. Often such smaller problems, such as sorting an array, may appear many times in different projects, and if you have solved the problem once, you should not have to do it again. These smaller problems can be developed in the form of a subroutine or a function, checked out, and then used in many different programs.

A simpler sort of repeatable procedure, which is not a true subprogram but which is an aid to the programmer, is the *statement function*. We began with this because it introduces the concept of passing *arguments* to procedures. A statement function is a single expression, in terms of constants and/or variables and *dummy variables*, which is given a name. It is of the form:

$$\text{fname (args)} = \text{expression (args)}$$

It saves the expression (sort of a formula to be evaluated) under the statement function name, and it can then be invoked to work on any appropriate set of values you wish to give it. The type of the statement function name determines the type of the value it will give when it is used. A statement function may make use of functions (system and user-defined) and of previously defined statement functions. An example to average any four real values:

$$\text{AVER (A, B, C, D)} = (\text{A} + \text{B} + \text{C} + \text{D}) / 4.0$$

Statement functions appear after declaration statements such as DIMENSION and type statements, and before any executable code.

A *subroutine* is a separately compiled, independent program unit, which can be called by another program unit, and can communicate with that program unit through an argument list or through the use of COMMON. A subroutine must begin with a line:

$$\text{SUBROUTINE name [(dummy arguments)]}$$

and it should include at least one RETURN statement (to take control back to the calling program), and terminate with an END statement. The subroutine may then be entered, from another program unit, by the use of a CALL statement:

$$\text{CALL name [(actual arguments)]}$$

The actual argument list in the CALL must agree in number and type with the dummy argument list for the subroutine. Dummy argument arrays must be dimensioned in the subroutine, using an \* or a dummy variable (or a constant). Two-dimensional arrays must have the number of rows specified (by

a dummy variable or a constant), but the number of columns can be left "assumed-length" (using an \*). A call to a sort subroutine might look like:

```
CALL SORT (CAT, 200)
...
SUBROUTINE SORT (X, N)
REAL X(N)
```

where the correlations between arguments are indicated.

The dummy arguments on the subroutine list simply allow the sharing of information with the calling program. Such information may be passed *in* to the subroutine for its use in calculations, or passed *out* of the subroutine back to the calling program, or it may go both ways ("*inout*"). Since there is no built-in protection in FORTRAN 77 to prevent accidental alteration of arguments passed to the subroutine, the programmer must exercise care.

A list of variables may be shared between program units by the use of COMMON (blank or labelled). A COMMON list sets up a special area in which variables are stored, and then can be referred to by any program unit with a corresponding COMMON list.

```
{Main program}
COMMON A, B, CAT(10) /RATS/ INK(20,30), BLUE
{Subroutine ONE} >
COMMON A, B, DOG(10)
{Subroutine TWO}
COMMON /RATS/ MINK(20,30), GREEN
```

In our example, the same variable may have different names (such as BLUE and GREEN) in different program units.

Variables in blank COMMON may not be initialized using DATA statements, but variables in labelled COMMON (/.../) may be initialized using DATA statements, in a BLOCK DATA subprogram. If character variables appear in an area of COMMON, no variables of any other type may appear in that same area.

A *function* operates much like a subroutine in most respects (argument lists, use of COMMON, RETURN, END, etc.), except that it stores a value into the name of the function before returning; this is its primary "function" (job). A function reference may appear in a complex expression in the program unit which invokes it. No subroutine or function in FORTRAN 77 may call itself.



## EXERCISES

1. Write a statement function that will calculate the effective resistance R of two resistors, whose values are R1 and R2, connected in parallel, according to the following equation:

$$\frac{1}{R} = \frac{1}{R1} + \frac{1}{R2}$$

2. Write a statement function (remember, a "one-liner") that will give the absolute value of any real argument X.
3. Write your own remainder or "mod" statement function, called MODULO, which will calculate the remainder when one integer argument is divided by another.
- 4. Use your "MODULO" statement function to evaluate whether any input year is a leap year, according to the following rules. A year is leap if it is divisible by 4, *unless* it is a century year (divisible by 100); a century year is only leap if it is also divisible by 400.
  - 5. Use your "MODULO" statement function to create a table of dates on which Easter Sunday falls, according to the following algorithm. For a given year, Y, calculate the following values: the remainder when Y is divided by 19 (call it A); the remainder when Y is divided by 4 (call it B); the remainder when Y is divided by 7 (call it C); the remainder when the quantity  $19A + 24$  is divided by 30 (call it D); and the remainder when the quantity  $2B + 4C + 6D + 5$  is divided by 7 (call this E). Then calculate  $22 + D + E$ ; this quantity is a date after the end of February; thus if it is  $\leq 31$ , it is in March, otherwise it is in April. Thus a value of  $22 + D + E$  of 35 is April 4. One other special test needs to be made; if the date in April turns out to be after April 25, subtract a week (that is, subtract 7). Use this algorithm to print out a table of all of the dates for Easter Sunday from 1901 through 2001, printing out "END OF THE DECADE" after every 10 years (i.e., after 1910, 1920, and so on), and "END OF THE CENTURY" after the year 2000.
  - 6. Write a statement function that will "round" a positive real argument up or down, as appropriate. If the fractional part of the real argument is 0.5 or greater, round up to the next integer; otherwise round down. Thus IROUND(3.7) should be 4, but IROUND(9.2) should be 9. Remember that you must do this as a "one-liner."
  - 7. Write a statement function to convert meters to feet, and another to convert feet to meters, if you know that 1 foot = 0.3048 meters.
  - 8. Write a statement function that will convert a number of yards, feet, and inches to total inches.
  - ♦ • 9. If a projectile is fired at an angle A (in radians) at an initial velocity  $V_0$ , write a statement function that will compute the maximum height of the projectile from the formula:

$$X_{MAX} = V_0^2 \sin A / G$$

where G is acceleration of gravity, 32.2 ft/sec<sup>2</sup>.

10. Write a subroutine which will accept any two filled integer arrays of any length and interchange their contents.
- 11. Write a subroutine which will determine whether any given character string is a *palindrome* or not (a palindrome reads the same backward and forward). Thus "MADAMIMADAM" (a contraction of "MADAM I'M ADAM") is a palindrome, but "HELLO MY NAME IS JOE" is not.
12. Write a subroutine which will examine two filled integer two-dimensional arrays of the same dimensions (the dimensions must also be provided in the call arguments), and determine if they are identical ('YES') or not ('NO').
13. Write a subroutine which will accept any positive integer value (within the limits of the machine word size) and break it up into its digits, storing each separate digit in a location of an array and then printing them out in reverse order.
14. Write a subroutine which will accept any positive integer value, determine its factors (including 1 but not the number itself) and sum up all of its factors.
15. Use the subroutine developed in the last exercise to do this problem. Two integers are said to be "amicable," or "friendly," if the sum of the factors of the first number is equal to the second number, and the sum of the factors of the second number is equal to the first number. Thus, 220 and 284 are amicable, since the factors of 220 are 1, 2, 4, 5, 10, 20, 22, 44, 55, and 110, which add up to 284; and the factors of 284 are 1, 2, 4, 71, and 142, which add up to 220. Write a subroutine which will accept any two positive integer values and print out whether they are amicable or not. Try 1184 and 1210.
16. You are beginning to write functional segments of what will be a *very* complex program to play chess. This program will have many subprograms, most of which need access to an  $8 \times 8$  array which represents the board. We will assume that integers are stored on the board to represent the pieces, and you have chosen different integers to represent each different piece—a 1 for the King, a 2 for the Queen, . . . , and a 6 for the Pawns. White pieces will simply have the appropriate integer for each type of piece; Black will represent each piece by adding 10 to the piece number, so a Black King will be 11, and so on. Write a FORTRAN subroutine, using COMMON for the board, which will set up the values for the initial board in the array.
- 17. Write a subroutine which will accept a two-dimensional array ( $5 \times 2$ ), or two one-dimensional arrays of length 5 each, which represent the positions (row, column) of five queens on an  $8 \times 8$  chessboard. A queen can "take" any piece in her row, her column, and on both diagonals through her position. Determine for the five queen positions given to your subroutine whether they "cover" the board (that is, whether a piece on any position of the board not already occupied by a queen could be taken by one of the queens); output whether all positions are covered.

**18.** (A more difficult chessboard problem). The “Eight Queens” Problem. The rules governing a chess queen’s domain of the board were given in the previous problem (17). Your program is to determine *all* possible configurations of the chessboard in which eight queens can be placed such that none of the queens can take any other queen. It is clear that there can only be one queen in each row, and one queen in each column. You should probably begin by placing queens, a row at a time, beginning in row 1. If your last placement makes it impossible to place the next queen anywhere on the board then “backtrack,” removing previous placements of queens, until you *can* place a queen in the row you are currently working on. There are 92 solutions to this problem; print them out.

**19.** Write a subroutine which will accept a pair of arrays of some specified size representing X and Y coordinates on a map. Have the subroutine determine all the distances between all pairs of points in the data, and print out which distance is greatest and the pair of points it lies between. Also print out the shortest distance and the pair of points it lies between. Make your code as efficient as possible; do not calculate a distance between points more than once; don’t compare a point with itself.

**20.** Write a function to find the largest value in any real one-dimensional array of any length.

- **21.** Write a function to find the smallest value in any integer two-dimensional array of any size.

**22.** Write an iterative function to calculate the cube root of any real argument.

**23.** Write an iterative function to calculate pi from one of the equations given in this chapter. Compare the accuracy of the different equations, and compare the accuracy you get by decreasing the value of your tolerance level epsilon. The value of pi to several places is:

3.1415926536

**24.** To do the iterative functions for sine, cosine, or  $e^x$  you need to calculate *factorials*. One definition of the factorial of N is the product of the integers from 1 to N. Using this definition, write a function that will calculate the factorial of an integer argument N (you should probably keep  $N \leq 12$  so as not to get overflow).

- **25.** *RECURSION.* A recursive function is one in which the operation is defined in terms of itself. For example, if there were no exponentiation operator in FORTRAN (as there is none in Pascal), exponentiation ( $x^{**n}$ ) could be defined recursively:

$$\text{power}(x,n) = x * \text{power}(x, n-1) \quad \text{power}(x,0) = 1$$

This definition assumes positive or zero values for the exponent n; also, since it cannot recurse infinitely, it has a defined *termination* value defined for  $n =$

0, that is,  $x^0 = 1$ . If FORTRAN allowed recursion (as will be the case in Fortran 90), this function could then be written (this is the Fortran 90 format):

```
RECURSIVE FUNCTION POWER (X,N) RESULT (XXX)
IF (N.EQ.0) THEN
    XXX = 1.0
ELSE
    XXX = X * POWER (X, N-1)
ENDIF
RETURN
END
```

There is also a *recursive* definition of  $n!$  ( $n$  factorial); it is:

$$n! = n \times (n-1)! \quad 0! = 1! = 1$$

A recursive definition is one that appears circular; it defines something in terms of itself. However, it is not really circular; it is (potentially infinitely) regressive. But each recursive definition should have a terminal point, a simplest case. For factorial, it is that  $0! = 1$ . Standard FORTRAN does not permit recursive definitions; that is, a subprogram may not call itself in the Standard. However, many FORTRAN compilers do permit recursion; determine if yours is one of these. Even if it is not, you will be able to use recursion in Fortran 90. Thus, write a recursive function for factorial.

**26.** Write a recursive function to find the  $n$ th Fibonacci number, if these are defined as follows:

$$F_n = F_{n-1} + F_{n-2} \quad F_1 = F_2 = 1$$

If you cannot run your recursive function on your system, write a straight iterative function to calculate the  $n$ th Fibonacci number. Which is easier to write?

**27.** Now that you have learned how to compute factorials, write an iterative function to calculate the sine of any angle in radians. Compare your results to those given by the system function SIN for the same arguments. Do not let overflow limit the number of terms you can use in your series; this time, to avoid overflow, make the factorials you compute reals.

**28.** Noting the similarities between the sine and cosine functions, write an iterative cosine function by modifying your sine function slightly. Compare to the system results.

**29.** Write an iterative function for  $e^x$ ; compare to the system function EXP(X).

**30.** The hyperbolic functions sinh and cosh are defined:

$$\sinh(x) = \frac{e^x - e^{-x}}{2} \quad \cosh(x) = \frac{e^x + e^{-x}}{2}$$

Write a function (not a statement function) that will calculate the hyperbolic sine of any real argument  $x$ , making use of the system function EXP. Note that  $a^{-b}$  is simply  $1/a^b$ , so do not use two calls to the function EXP, which would be unnecessarily time-consuming. Now modify your SINH function to create a function that will calculate a hyperbolic cosine (COSH).

**31.** *Greatest Common Divisor.* Euclid's algorithm for finding the greatest common divisor of any two nonzero integers was explained in Chapter 3. Write an integer function which will take the greatest common divisor of any two integer arguments according to this algorithm. Be careful to *protect* the arguments passed to the function (that is, do not change them in the function), since their values should not be altered when you return to the calling program. Do this by "copying" them into local variables in the function before you begin implementing the algorithm. Try it out on several values. Notice this might be useful if your publisher reported to you the dollar amount of total sales of your book in two different 3-month periods, and did not tell you the selling price of the book. Because you want to know how many copies of the book were sold, you find the greatest common divisor of the two dollar totals; since you know roughly the cost range of the book, you should be able to determine how many copies were sold.

**32.** *Least Common Multiple.* The Least Common Multiple (LCM) of two integers is the smallest integer into which they will both divide exactly. A little thought will tell you that this is not necessarily just the product of the two numbers (unless they are relatively prime to one another). For example, the least common multiple of 20 and 24 is 120 (not 480); this is because they have the common factor 4. The LCM can be determined by using the greatest common divisor (GCD) of the two integers:

$$\text{LCM}(M,N) = M * N / \text{GCD}(M,N)$$

Write a statement function which will find the LCM of two integer arguments, making use of your Greatest Common Divisor function, developed in the previous exercise. Use this to determine, for example, the smallest floor area you can tile exactly with tiles that are *either* 6" x 6", or 10" x 10".

- **33.** *Alternate ENTRY.* A feature available for subprograms, which we did not discuss in the main chapter is that of creating alternate entry points to a subroutine or function. The primary entry point is the FUNCTION or SUBROUTINE line beginning the subprogram. Additional entry points can be created by inserting lines of this type in the subprogram:

ENTRY altname [dummy arguments]

Then, if the alternate subprogram name is used, execution of the subprogram will begin immediately following that entry line. This can be useful if there is a great similarity in the internal operations of two or more different subprograms.

Consider, for example, the great similarity in the two iterative formulas for square root and cube root discussed earlier in this chapter. To write a CUBRT function, instead of rewriting most of the same code used for SQROOT, modify the SQROOT function to have an alternate entry point if you want to calculate a cube root, setting up the appropriate different parameters in each case.

All of the entry point names are equated if you use multiple entries. Thus if you have entry points SQROOT and CUBRT in your function, the answer can be stored in either name, no matter what entry point you took to the function. This saves additional testing to determine which name the result should be stored in.

**34.** Use the alternate ENTRY technique discussed in the preceding problem to write one function with two different entry points, to calculate either a sine or a cosine of an angle given in radians, using iteration on the infinite series expressions for those given in the chapter. Notice how greatly similar the two series are, and make use of this in your function. Use loops to test your sine and cosine functions against the system functions SIN and COS for a variety of arguments.

- **35.** *Alternate RETURNS.* A subroutine (not a function) may be entered with a calling sequence that specifies that, under certain conditions in the subroutine, different "returns" may be made to distinct points in the calling program. This is done by specifying a number of statement numbers, each preceded by an \*, in the calling sequence, and a corresponding set of \*'s in the dummy argument list of the subroutine entry line; for example,

```
CALL TRICKY (A, B, *3, *5, KANT, *2)
...
SUBROUTINE TRICKY (X, Y, *, *, KK, *)
```

Each \* in the subroutine entry line corresponds to a starred statement number in the calling statement; if the instruction RETURN 1 is executed in the subroutine, after re-entering the calling program the statement number associated with the first \* in the CALL statement will be executed (in our example, statement number 3); if the statement RETURN 2 is executed in the subroutine, then the return will be to the statement number in the calling program associated with the second \* in the CALL (in our example, statement 5); and so on. If a simple RETURN, without an integer following it, is executed in the subroutine, a normal return (to the first executable statement following the CALL) will be executed. This feature allows conditions determined in the subroutine to affect which statement is to be executed after the subroutine has done its work.

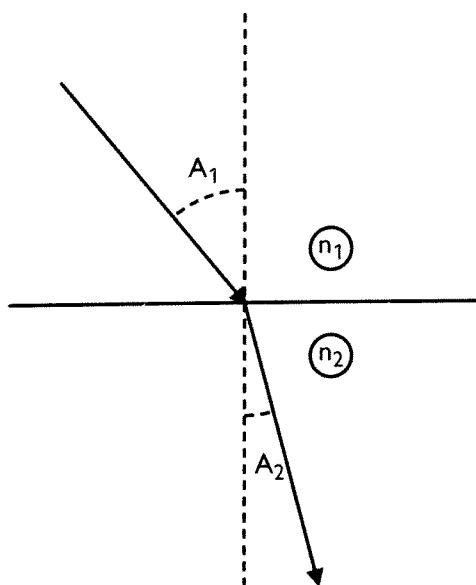
Write a FORTRAN program which uses a subroutine to evaluate a tic-tac-toe board (a  $3 \times 3$  array), which has 0s and 1s in the played positions, and 9s in the unplayed (blank) positions; the subroutine is to evaluate and print out

whether the board shows a WIN for 0, in which case the RETURN should be to statement 10 in the calling program; or a WIN for 1, in which case the RETURN should be to statement 20 in the calling program; or a DRAW, in which case no more plays can be made (filled board), and a RETURN should come back to statement 30 of the calling program; or it is a PLAYON situation, in which case a normal RETURN should be executed. Your main program should set up the board values sent to the subroutine; in a more interesting version, you might have the main program allow two players to make alternate plays on the board until either one wins or a draw situation arises. After completing Chapter 13 on simulations and models, you can write a program to have the computer make random moves against you. An even more interesting program is to write strategy for the computer to make plays against the human opponent. You may work out a strategy for the computer, or refer to problem 21 in Edgar, *Advanced Problem Solving with FORTRAN 77* (SRA, 1989), pp. 172–173.

**36.** The Twelve Knights Problem. This is a variation on problem 17, the “five queens” problem. Place (randomly—see Chapter 13—or by reading in) 12 knights on a chessboard. A knight can move in an “L” in any direction, and the “L” may be one space up and two over, or two spaces up and one over (and the “over” direction can be either of those two directions perpendicular to the first move). Determine whether your 12 knights “cover” the board completely or not.

**37.** The index of refraction of a substance,  $n$ , is the ratio of the velocity of light in a vacuum (about  $3 \times 10^8$  m/sec) to its velocity in that medium. For example, the index of refraction of water is 1.333, of quartz 1.544, etc. Write a function which will accept an index of refraction and return the velocity of light in the medium that index represents.

**38.** *Snell's Law.* If light passes from one medium to another, Snell's law tells us that the relation between the angle of incidence of the incident ray and that of the refracted ray is:



$$n_1 \sin A_1 = n_2 \sin A_2$$

where  $n_1$  and  $n_2$  are the indices of refraction of the two media (see the previous problem), and the  $A$ 's are the respective angles. Write a function that will accept two values for the index of refraction for two different media and the angle of incidence of light in the first, and calculate the angle of refraction in the second medium.

# CHAPTER 10



## ADDITIONAL CONTROL STATEMENTS AND DATA MANIPULATION

There are several additional kinds of useful control statements that can be implemented in a programming language, and the set of those in FORTRAN is assembled here. Many very interesting problems are nonnumeric, and so require special character manipulation, which can be performed in FORTRAN. Small values can be *packed* several to a computer word, and large values can be spread over many locations (in an array). Techniques to accomplish these and other sleights-of-hand are introduced in this chapter.

The Rosetta stone, an ancient Egyptian slab of basalt inscribed in hieroglyphic, demotic (a later form of hieroglyphic), and Greek, was used by the French Egyptologist Champollion to decipher the language of hieroglyphics in 1821.

*"He had forty-two boxes, all carefully packed, With his name painted clearly on each;  
But, since he omitted to mention the fact, They were all left behind on the beach."*

- Lewis Carroll, Hunting of the Snark

## ◆ THE ELSEIF STRUCTURE FOR CASE

Some languages incorporate a “case” statement to allow the testing of a variable or expression and, for different possible values of that expression, to execute various different sections of code. A general pseudocode form of such a structure is:

```
CASE expression of
    value(s)1: code1;
    value(s)2: code2;
    ...
    end
```

The expression is evaluated, and compared to the value or values in each one of the cases listed until a match is found and the appropriate code is executed. If it does not match anywhere, it may cause an error condition, though on some systems the syntax may allow an “otherwise” case to which it can default.

In Fortran 90, there will be a SELECT CASE structure of the form:

```
SELECT CASE (case-expression)
CASE case-selector1
    action1
CASE case-selector2
    action2
    ...
    [ CASE (DEFAULT) ]           {optional}
END SELECT
```

Each of the case-expressions can be of one of the following forms:

(low: high)	(low:)
(:high)	
or	(DEFAULT)

Thus, for example, in Fortran 90, one would be able to write a CASE program segment to select on numerical grades to turn them into letter equivalents, or convert wavelength ( $\times 10^{-7}$ ) to color:

```
SELECT CASE (GRADE)
    (90: ) PRINT*, 'A'
    (80:90) PRINT*, 'B'
```

```
SELECT CASE (COLOR)
    (4.0: 4.6) PRINT*, 'VIOLET'
    (4.6: 4.9) PRINT*, 'BLUE'
```

```

(70:80) PRINT*, 'C'
(60:70) PRINT*, 'D'
(DEFAULT) PRINT*, 'E'
END SELECT
(4.9: 5.7) PRINT*, 'GREEN'
(5.7: 5.95) PRINT*, 'YELLOW'
(5.95: 6.2) PRINT*, 'ORANGE'
(6.2: 7.0) PRINT*, 'RED'
END SELECT

```

Until this innovation becomes available in Fortran 90 compilers, the FORTRAN 77 programmer will have to make do with a Block IF/THEN/ELSEIF structure (or perhaps a computed GO TO, to be discussed next) to handle such "case" problems. The preceding example regarding grade ranges could easily be written:

```

IF (GRADE.GE.90) THEN
  PRINT*, 'A'
ELSEIF (GRADE.GE.80.AND.GRADE.LT.90) THEN
  {or ELSEIF (GRADE. GE. 80) THEN }
  PRINT*, 'B'
ELSEIF (GRADE.GE.70) THEN
  PRINT*, 'C'
ELSEIF (GRADE.GE.60) THEN
  PRINT*, 'D'
ELSE
  PRINT*, 'E'
ENDIF

```

and similar tests for the wavelengths. Note that an ELSEIF can implement the (low:high) case range using the form:

```
ELSEIF (expression.GE.low .AND. expression.LE.high) THEN
```

Because of the sequential nature of our test cases, we really only had to test whether the expression was greater than or equal to each new lower bound, and we did that on subsequent tests. We implemented the CASE (DEFAULT) with an ELSE, and in the color problem, if no test was satisfied, nothing was printed at all.

In this manner, we can easily take care of any set of conditions that would be handled by a "case" structure. The addition of a SELECT CASE structure to Fortran 90 is largely to make such an option available to programmers coming to Fortran from a language such as Pascal, where they have become accustomed to having such a structure available.

Notice that in a Block IF/THEN/ELSEIF, once a condition has been satisfied, none of the other conditions has to be tested, and the program, after executing the indicated expression(s), drops down to the point following the ENDIF. This is much more efficient than having several successive IF/THEN blocks to run. The case ranges in the preceding examples were contiguous, but that need not necessarily be so. Imagine the case where you want to modify measurements that fell in two particular ranges, but not elsewhere. Measurements in the

range 0.5 to 0.75 you want to increase by .05, and measurements that fell in the range from 1.6 through 1.8 you want to decrease by 10%. You could do this with a Fortran 90 SELECT CASE structure, if available, or with a FORTRAN 77 IF/THEN/ELSEIF block. You would not, however, be able to do it with most Pascal CASE implementations, for two reasons: (1) the Pascal CASE statement accepts a value or a group of values, separated by commas, to identify a case, but generally not a range of values (thus, Fortran is more flexible); (2) if a value does not fit one of the CASE conditions in the structure, an execution error may occur. In Fortran, on the other hand, if the value of the case expression to be tested does not match any of the conditions set up, the program will simply pick up execution after the test structure (SELECT CASE or IF/THEN).

```

READ*, VALUE
IF (VALUE.GE.0.5 .AND. VALUE.LE.0.75) THEN
    VALUE = VALUE + 0.05
ELSEIF (VALUE.GE.1.6 .AND. VALUE.LE.1.8) THEN
    VALUE = 0.9*VALUE
ENDIF

```

All other values, not in one of the two specified ranges, will remain unchanged.



## THE COMPUTED GO TO

The unconditional branch, or GO TO, is rightly deprecated in other languages, but in FORTRAN it still is necessary to allow for the premature exit from a loop (for example if an error or other special condition occurs), and to allow the programmer to simulate WHILE/DO and REPEAT/UNTIL structures (as we saw in Chapter 4), until these structures become standardly available in the language. Otherwise it should be avoided.

Another variant on the GO TO statement in FORTRAN is the computed GO TO, one in which the branch taken depends on the value of an integer variable or expression. Instead of the simple unconditional branch,

GO TO n

which must transfer control to statement n, no questions asked, the computed GO TO is of the form:

GO TO (n<sub>1</sub>, n<sub>2</sub>, ..., n<sub>m</sub>)[,] intexp

Following the GO TO are a number (m) of different statement labels in parentheses, separated by commas. After the parentheses, and an optional comma, is an integer expression (a variable or more complex expression to be

evaluated). The value of the integer expression is expected to lie in the integer range from 1 through m, and its value determines which branch is to be taken. If the value is 1, the branch to the first statement in the list,  $n_1$ , is taken; if the value is 2, the branch to the second statement number in the list,  $n_2$ , is taken; and so on. If the value of *intexp* does not fall in the range from 1 to m, the next expression following the computed GO TO is executed.

Thus, for example, in the instruction

```
GO TO (3, 7, 8, 5, 7) MIX
PRINT*, MIX
```

statements labelled 3, 7, 8, and 5 must exist in the program. If the value of the variable MIX is 1, a branch will be taken to statement 3; if MIX is 2 or 5, a branch will be taken to statement 7; if MIX is 3, a branch will be taken to statement 8; and if MIX has the value 4, a branch will be taken to statement 5. If MIX is less than 1 or greater than 5, the statement following the GO TO (that is, PRINT\*, MIX) will be executed.

A computed GO TO might be especially useful if you have set a "flag" or "switch" value in your program, that does different operations in each pass through a loop. Let us say that your program is to generate a sequence of numbers according to the following rules: (1) begin with some integer value; (2) the next value comes from the previous one by adding 3; (3) the next value is the last one times 2; (4) the next value is the last one minus 4; (5) repeat from step (2). If you select an initial value, the following loop will generate the next 100 values in the sequence, according to the rules, by using a "switch" value and a computed GO TO statement.

```
INTEGER SWITCH, INIT, NEXT
READ*, INIT
PRINT*, INIT
NEXT = INIT
SWITCH = 1
DO 30 I = 1, 100
    GO TO (10, 15, 20) SWITCH
10    NEXT = NEXT + 3
        SWITCH = 2
        GO TO 25
15    NEXT = NEXT*2
        SWITCH = 3
        GO TO 25
20    NEXT = NEXT - 4
        SWITCH = 1
25    PRINT*, NEXT
30    CONTINUE
```

If the sequence were begun with the initial value 5, it would proceed as follows:

5, 8, 16, 12, 15, 30, 26, and so on.

Note that the same job could be accomplished using a block IF/THEN/ELSEIF structure:

```

READ*, INIT
PRINT*, INIT
NEXT = INIT
DO 30 I = 1, 100
    IF (SWITCH.EQ.1) THEN
        NEXT = NEXT + 3
        SWITCH = 2
    ELSEIF (SWITCH.EQ.2) THEN
        NEXT = NEXT*2
        SWITCH = 3
    ELSE
        NEXT = NEXT - 4
        SWITCH = 1
    ENDIF
    PRINT*, NEXT
30 CONTINUE

```

Many programmers find the IF/THEN structure preferable. However, you should be familiar with the computed GO TO in case you encounter it in programs you have to read or maintain.

If one is analysing a program containing numbered answers to a questionnaire that fall in a predictable range, and the action to be taken depends on the answer given, a computed GO TO may be helpful. A programmer is writing a Computer Dating program, in which answers are used to compute a weighted score of one person's answers to questions matched against the questionnaires filled out by those of the opposite sex. We will make up a facetious question and possible selection of answers, plus rules for scoring these answers:

#### What is your favorite flavor of ice cream?

---

- |                |               |                  |                |
|----------------|---------------|------------------|----------------|
| (1) vanilla    | (2) chocolate | (3) pistachio    | (4) rocky road |
| (5) rum raisin | (6) coffee    | (7) butterscotch | (8) strawberry |

This question is to be scored as follows: if the respondent answers vanilla (1), add 1 to the score so far; if the answer is chocolate (2) or strawberry (8), add 2; if the answer is coffee (6) or butterscotch (7), add 3; if the answer is rocky road (4) or rum raisin (5), add 4; if the answer is pistachio (3), subtract 3, unless the person against whom these answers are being compared also answered

pistachio, in which case, add 5. If the answer to this question by a particular possible match of the opposite sex is stored in location ICE, and the answer given by the person for whom you are computing the scores of possible matches is stored in MYICE, the program could be written:

```

INTEGER SCORE
. . . {scoring program up to this point}
GO TO (10, 20, 30, 40, 40, 50, 50, 20) ICE
10 SCORE = SCORE + 1
    GO TO 60
20 SCORE = SCORE + 2
    GO TO 60
30 IF (MYICE.EQ.3) THEN
    SCORE = SCORE + 5
ELSE
    SCORE = SCORE - 3
ENDIF
    GO TO 60
40 SCORE = SCORE + 4
    GO TO 60
50 SCORE = SCORE + 3
60 . . . { next response is analysed}

```

We could also have used the computed GO TO to handle the problem in the previous section in which we assigned letter grades according to a range of numerical grades. Instead of using a "case" structure or a preferable IF/THEN/ELSEIF, we could write:

```

GO TO (100, 120, 140, 160), 10 - INT(GRADE)/10
PRINT*, 'E'
    GO TO 200
100 PRINT*, 'A'
    GO TO 200
120 PRINT*, 'B'
    GO TO 200
140 PRINT*, 'C'
    GO TO 200
160 PRINT*, 'D'
200 . . .

```

This program assumes that no one got a grade of 100 or above; if they did, what letter grade would they receive, according to this program's code?!

The previous example illustrates that the use of a computed GO TO engenders the need for more GO TOs, in the many-condition examples we have

used. It is for just such a reason that the computed GO TO is generally not used in modern programming circles. However, you may encounter it in earlier programs you will have to update and maintain.

## ASSIGNED GO TO

Though the Assigned GO TO is not much in use, we include it here for the sake of completeness. The ASSIGN statement may be used to assign a statement number value to an integer variable:

ASSIGN n TO i

with the result that the statement number n (a constant, which must represent an actual statement number existing in the program) is given to the variable i. This assigned variable may then be used to indicate a FORMAT statement in an I/O instruction (for example, PRINT i, list), or it may be referenced in an assigned GO TO.

An Assigned GO TO statement may be of the form:

GO TO i

or

GO TO i [ (s<sub>1</sub> [s<sub>2</sub>, . . .] ) ]

and it indicates an unconditional branch to the statement number of the value i (and i *must* have been defined using an ASSIGN statement). If the optional statement number(s) (s<sub>1</sub>, etc.) are used in parentheses after the assigned GO TO, they may represent a list of the statement numbers this GO TO may branch to, and then i must be one of the s-values on the list.

This construction has very limited value, but at least you will have seen it if it occurs in some older program you are given to maintain and update.

## EQUIVALENCE

There are several uses for EQUIVALENCE, which is a way to equate areas of memory to one another. This feature allows the programmer to have several different names for the same location in memory. One possible use of this is to make a quick "patch" in a program if you have inadvertently called a variable by different names throughout the program. This might occur because segments of the program were written at different times, when your mood was different (such as PARTY, WORK, DRINK, and SONG). To make all of these names refer to the *same* location in memory, one would use the statement:

## EQUIVALENCE (PARTY, WORK, DRINK, SONG)

A look at a symbolic reference map for this program would indicate all of these different variable names referring to the same program address. Another type of error is the *typo*, which is possibly just a transposition in letters in a variable name, such as occasionally typing TERM as TREM. This can be fixed by:

## EQUIVALENCE (TERM, TREM)

However, such quick patches may allow you to quickly remove some program errors in order to unearth other, more subtle, ones, but they should not be left in a final version of a program. It is potentially confusing to a reader of your program (even yourself, at some later date) to have such equivalences there just to cover up for organizational or typing errors. Today's modern screen text editors make it relatively simple to "find" and "fix" such errors by appropriate replacements of strings by other strings.

There are other interesting uses of EQUIVALENCE. In earlier systems, there was often a problem having enough memory to run a program that handled large amounts of data. This is less of a problem on computers today, except for a few special programs that require huge amounts of memory, but it still may be a problem in running programs on smaller microcomputers. The use of EQUIVALENCE allows the programmer to equate large areas of memory. Thus, if you had a program that dealt with an array A of 250,000 locations early in the program, and another array B of 200,000 locations later in the program, that would take up 450,000 spaces in memory, which might be pushing your capacity. However, if you are all finished with the array A by the time you come to the part of the program that deals with array B, you could allow B to share the space that had been allocated to A:

```
REAL A(250000), B(200000)
EQUIVALENCE ( A(1), B(1) )
```

Notice that to equate the 200,000 locations of array B to the first 200,000 locations in A, all that is required is to say that their first locations ( A(1) and B(1) ) are equivalent. Since arrays are stored consecutively in memory, if A(1) lines up with B(1), then A(2) lines up with B(2), and so on. Now this program only requires 250,000 spaces for the arrays, instead of 450,000.

You might say that this could have been accomplished without the use of EQUIVALENCE, simply by having *one* array A, dimensioned to 250,000 locations, and using part of it to do the job required for array B later in the program. This is true in this example, although it might be easier for someone reading the program to understand what was going on if the two arrays had distinct names, since they are doing distinct jobs. Since both arrays are one-dimensional, and of type real, we *could* simply use the same name for both, and then

just use extensive comments in the program to explain which job the array was doing at a particular point in the program.

The approach of re-using the same array name to do two jobs would work in the previous example, but it would not work if the two arrays in question were of different types (say, one integer and the other real), or if they were of different dimensionality. By using EQUIVALENCE we could equate a real one-dimensional array to an integer one-dimensional array, and also to a real (or logical, etc.) two-dimensional array. For example,

```
REAL C(2000), D(40, 50)
INTEGER KAT(2000)
EQUIVALENCE ( C(1), D(1,1), KAT(1) )
```

would allow these three arrays to share the same 2000 locations of memory space. This could not be handled simply by allowing the array C to do different jobs at different parts of the program. Thus, EQUIVALENCE allows locations in memory to be treated as different *types* for various purposes (with the exception, in FORTRAN 77, that a CHARACTER type location cannot be equated to a variable of a different type, since the character storage location is not standardized from one machine implementation to another).

## Using EQUIVALENCE to Equate Different Types of Values

The capability to treat a location as having different types allows the programmer to see what a real value would look like if handled as an integer, and vice versa. For example,

```
INTEGER NEXT
REAL A
EQUIVALENCE (NEXT, A)
A = 3.75
PRINT 8, NEXT
8 FORMAT(5X, I15)
NEXT = 75
PRINT 9, A
9 FORMAT(5X, G12.3)
```

would allow you to experiment and see why it is important to treat real values as reals, and integers as integers. It would emphasize the importance of having an argument to a subprogram treated as the same type in both the calling program and the subprogram, or else great distortions in value (as indicated by this example) could occur.

This flexibility could also allow the programmer to treat a location alternately as an integer, say, and then as a logical, allowing logical operations to be performed on a location that had been initially defined while treating it as an integer. This would allow “packing” binary 1 and 0 values into a location by treating it as an integer, and then performing logical AND, OR, and NOT operations on it when it is treated as the equivalent logical. As a simple example, the integer value 12 has the binary value  $1100_2$  and the integer value 10 has the binary value  $1010_2$ . These could be stored in two integer locations, which were then equated to logical variables; they could then be considered to represent the truth values TTFF and TFTF, respectively, and then be used to determine the “truth table” for AND and OR, given these values. For example,

```

INTEGER IX, IY, IZ
LOGICAL X, Y, Z
EQUIVALENCE (IX, X), (IY, Y), (IZ, Z)
IX = 12
IY = 10
Z = X.AND.Y
PRINT*, IZ
Z = X.OR.Y
PRINT*, IZ

```

The AND operation on 1100 and 1010 should give the logical bit pattern 1000, so IZ would print out the integer value 8. The OR operation on 1100 and 1010 should give the logical bit pattern 1110, so IZ would print out the integer equivalent value 14. The conversions from decimal to binary and binary to decimal are discussed in Appendix A, Number Systems. The individual logical bit values could be pulled out from the integer location IZ by appropriate operations. For example, the leftmost bit value in IZ, if it represents a four-bit string, would be  $IZ/8$ ; the next would be  $(IZ - IZ/8*8)/4$ ; the next would be  $(IZ - IZ/4*4)/2$ ; and the last would be  $MOD(IZ, 2)$ . These extracted bit values could then be converted to a T/F string.

The AND and OR operations can also be used when talking about membership in sets, as we shall see in a later section. The AND relation applied to two sets gives the *intersection* of the sets, and the OR relation gives the *union*.

## Using EQUIVALENCE to Equate Arrays of Different Dimensions

Arrays of different dimensions can also be equivalenced. If we have a multi-dimensional array (say, for example, a three-dimensional array) and we want to do the same operation to *every* element of the array, independent of order, we can use EQUIVALENCE to do it more efficiently. For example, imagine you

## 376 ADDITIONAL CONTROL STATEMENTS AND DATA MANIPULATION

have a three-dimensional integer array of 10 rows, 20 columns, and 30 "pages." It has already been filled with values, and you simply want to add up all of the values so that you can compute the average of the values in the array. You would normally do this the following way:

```
INTEGER MARK(10, 20, 30)
. . . {values read into MARK}
MSUM = 0
DO 10 I = 1, 10
    DO 10 J = 1, 20
        DO 10 K = 1, 30
            MSUM = MSUM + MARK(I, J, K)
10 CONTINUE
SUM = MSUM
AVER = SUM/6000
```

This example clearly shows what it is doing, but it is very inefficient. Loops have a high "overhead" cost, of initializing, incrementing, and testing, and here there are *three* nested loops. Furthermore, in FORTRAN 77 there is the additional overhead of pre-checking each DO loop to see if it has a positive number of iterations (this was discussed in Chapter 4). Since the loops access each element of the array, and since the order in which this is done does not matter, it would be much simpler if we could accomplish the task with *one* loop instead of three. We can do this if we equivalence the three-dimensional array to an integer one-dimensional array.

The order in which the three-dimensional array is stored in memory is "page" by "page," and within each page, column by column (as we saw with two-dimensional arrays). If we process the one-dimensional array in order, it will be equivalent to nesting the three loops of the example in this alternate order:

```
DO 10 K = 1, 30
    DO 10 J = 1, 20
        DO 10 I = 1, 10
```

but we see that this has no adverse effect on the result. We can thus accomplish this job much more quickly by:

```
INTEGER MARK (10, 20, 30), LARK (6000)
EQUIVALENCE ( MARK(1,1,1), LARK(1) )
MSUM = 0
DO 20 I = 1, 6000
    MSUM = MSUM + LARK(I)
20 CONTINUE
SUM = MSUM
AVER = SUM/6000
```

This sort of equivalencing of an array of higher dimensionality to a one-dimensional array is effective for any operation in which all elements of the large array are to be treated equally—for example, finding a maximum or a minimum, normalizing all the values in the array, or replacing each value in the array by its square.

## Using EQUIVALENCE With COMMON

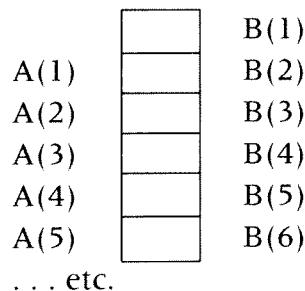
Mappings of variables onto other variables (or arrays) using EQUIVALENCE has a few limitations. Character variables can *only* be mapped onto other character variables. Array mappings must be *consistent*; that is, if we equate A(1) with B(2), as in the following example:

```
REAL A(20), B(25)
EQUIVALENCE ( A(1), B(2) )
```

then it would be *inconsistent* to also map A(5) onto B(4) with a statement such as:

EQUIVALENCE ( A(5), B(4) )	{illegal; inconsistent}
----------------------------	-------------------------

since, if A(1) maps onto B(2), the arrays line up as follows:



so the first equivalence lines A(1) up with B(2), making all of the rest of the two arrays line up accordingly. This makes A(5) equivalent to B(6), and so it cannot also be equivalent to B(4), as the second, inconsistent, EQUIVALENCE statement would try to make it.

Another restriction on the use of EQUIVALENCE is in conjunction with COMMON. Since a COMMON declaration establishes a specific order in which variables will be stored in a special area of memory, an EQUIVALENCE statement cannot attempt to alter this order. Thus, if we have the statements:

```
COMMON A, B, KATE
COMMON /FATE/ X, Y, Z
```

this establishes that A, B, and KATE are the first three locations, *in that order*,

in blank COMMON, and the variable X is the *first* location in the labelled COMMON area FATE, Y is the *second* location in that area, and Z is the third. Thus, we could not equivalence A to KATE or B to Y

```
EQUIVALENCE (A, KATE), (B, Y) {illegal}
```

since A cannot be both the first and the third location in blank COMMON, and B cannot be both the second location in blank COMMON and also the second location in the COMMON area labelled FATE. Thus, the general rule is, you cannot equivalence *any* two variables in areas of COMMON.

An additional restriction is that, if you equivalence a variable in COMMON to one not in COMMON (which effectively puts the latter variable into COMMON), you may do so as long as you do not extend the COMMON area upward. Thus,

```
DIMENSION X(5)
COMMON A(4)
EQUIVALENCE (A(2), X(1))
```

is legal, places array X into COMMON, and extends COMMON downward as indicated:

A(1)	
A(2)	X(1)
A(3)	X(2)
A(4)	X(3)
	X(4)
	X(5)

but the following would not be legal:

```
DIMENSION Y(5)
COMMON B(4)
EQUIVALENCE (B(1), Y(2)) {illegal}
```

since it would extend COMMON upward, as shown:

	Y(1)
B(1)	Y(2)
B(2)	Y(3)
B(3)	Y(4)
B(4)	Y(5)

This is illegal because blank COMMON, as shown here, is stored in memory where the loader was resident, immediately below the program area itself. If blank COMMON is extended *upward*, it will encroach on locations in the program itself, and perhaps wipe out an important value, or even an instruction.

Even labelled areas of COMMON cannot legally be extended upward through the use of EQUIVALENCE. Presumably they are already laid out in a specific mapping onto memory, relative to one another and the rest of the program, and going above the top of a labelled area of COMMON might destroy other information, possibly even in the resident monitor or the program area being used by another person on a multi-user system.



## CHARACTER DATA AND TEXT HANDLING

FORTRAN 77 has greatly enhanced its capabilities for handling nonnumeric data, that is, text or character data. There were serious restrictions on what earlier versions of FORTRAN could do with character strings, and great incompatibilities from one installation to another regarding how to accomplish any character storage and manipulation. FORTRAN 77 has removed these problems by creating the CHARACTER data type, while leaving the details of the implementation of its storage to the individual installations and the compilers they install.

There are many applications which require manipulation of nonnumeric, symbolic data, even for the scientist and engineer. We shall thus take a look in this section on the character-handling capabilities of FORTRAN 77, and their potential for various interesting applications.

As we have already seen, in FORTRAN 77 you declare variables which are to contain character strings to be of type CHARACTER, and to be of a specified length (default length is 1). These character strings have no specified length limit in the Standard, but particular implementations may impose a limitation (such as 255 or 256). The way in which these are actually stored in memory will vary from one installation to another, but the programmer generally does not need to worry about this. From the FORTRAN level, these strings can be viewed as simply occupying one (large) location, since each string has its own unique single name. As discussed in Chapter 5, substrings can be extracted from character strings, and character strings can be *concatenated* with one another to form longer strings. Review this material if you have forgotten it, because it will be useful here.

Character strings may be single variables, or elements in an array, as indicated:

```
CHARACTER*5 CAT, DOG(100), CALEND(6,7)*2
```

Values can be given to character variables using an assignment statement, a READ, or a DATA statement. Character values can be *concatenated* (that is, strung

together), taken apart (using the substring notation), or compared to one another (which can indicate equality, inequality, or order in the collating sequence of the machine you are using and/or in the ASCII sequence; for the latter, use the logical lexical functions LLE, LLT, and so forth that are discussed in Appendix B).

In Chapter 5, we also discussed the CHAR and ICHAR functions. Let us quickly remind you how these functions operate. The CHAR function takes an integer argument, which is a position in the collating sequence, and returns the character that is found in that position. The ICHAR function takes a character argument and returns the integer position of that character in the collating sequence. The use of these two system functions in tandem allows nice manipulations such as easily generating the letters of the alphabet (uppercase or lowercase), turning a digit into its character equivalent, and vice versa. For example, imagine that *you* had to take the character string entered for an integer value (since all formatted I/O is actually done in terms of character strings) and convert it to its equivalent integer value. We will assume the "blanks are zeros" convention, so that leading blanks in the field are ignored but trailing blanks will be interpreted as zeros.

Let us assume the value is read in with an A6 format. Thus you will have a 6-character string which must be reinterpreted as an integer. First, we will go by any leading blanks, and as soon as we hit a nonblank character, we will convert it to its equivalent integer digit. Then it must be multiplied by the appropriate power of ten and added into the number we are building according to this process. We saw in Chapter 5 how to convert a character digit into an equivalent integer—imply subtract ICHAR('0') from the position of your character (C) in the collating sequence (that is, ICHAR(C)).

```
***** CONVERT AN N-CHARACTER STRING TO AN INTEGER *****
***** ASSUMING BZ CONVENTION (BLANKS ARE ZEROS) *****

PARAMETER (N = 6)
CHARACTER IN*N, C
INTEGER NUMB, ZERORF
ZERORF = ICHAR('0')
READ 4, IN
4 FORMAT(A6)
I = 1
5 IF ( IN(I:I).EQ.' ' .AND. I.LT.N) THEN
    I = I + 1
    GO TO 5
ENDIF
NUMB = 0
IF (I.LE.N .AND. IN(N:N).NE.' ') THEN
    MULT = 10** (N-I)
    DO 10 J = I, N
        C = IN(J:J)
```

```

    IF (C. NE. ' ') THEN
        K = ICHAR(C) - ZERORF
        NUMB = NUMB + K*MULT
    ENDIF
    MULT = MULT/10
10    CONTINUE
ENDIF
{now you can do calculations with NUMB, etc.}

```

It would be much more interesting to have to convert a character string to an equivalent *real* value!



## PACKING SMALL VALUES

Occasionally the need arises to use a single memory location to contain more than one value (matters of limited memory, or just bringing in and out fewer data items in I/O, which is the slowest part of the program operation, or to perform complex logical comparisons). For example, if you were dealing with a great many integer values that lay in the range from 0 through 21 and wanted to "pack" them into computer words, even on a 32-bit machine you could pack five into each word. Let us imagine a problem in which these values are being read in or generated by computation, say 100,000 of them. We can then pack them into an integer array of only 20,000 locations, with something like the following program. The array could then be written out to disk.

To think a bit about our procedure first, we will pack values from right to left in each memory word (SAVE) until it is filled. We will do this by multiplying the first value by 1, the next by 100, the next by 10,000, and so on, adding them all into SAVE until five values have been packed there; then we will store the value into the next location of the array (PACKED). The counter K will keep track of where we are filling the array.

```

PROGRAM PACKUP
INTEGER PACKED(20000), AGE, SAVE, MULT
K = 0
SAVE = 0
***** MULT IS A POWER OF 10, DETERMINES WHERE TO PACK AGE *****
MULT = 1
DO 99 I = 1, 100000
    {logic which reads in or calculates integer value AGE}
    SAVE = SAVE + AGE*MULT
    IF (MULT.LE.1000000) THEN
        MULT = MULT*100
    ENDIF
99

```

```

    ELSE
        K = K + 1
        PACKED(K) = SAVE
        SAVE = 0
        MULT = 1
    ENDIF
99  CONTINUE
    {write out array to disk}
STOP
END

```

Now think about how you would write another program which would read in the array PACKED, and *unpack* the values stored there so that they could be processed, one at a time. Recall that each value is stored in a two-digit "slot," and that they can be "peeled off" by using something like the MOD function. Remove the values from the array location from right to left, so that they are used in the same order in which they were generated by the first program, before packing.

## Packing True/False Values for Logical Manipulation

If we are dealing with logical values, they can only be either .TRUE. or .FALSE. (or 1 or 0). In normal memory storage, however, a whole word is used for each logical value, wasting 31 (or 47 or 59 or 63) bits in the rest of the word. We could "pack" such binary logical values into single words in memory by multiplying them (as 1 or 0 values) by successive powers of two and then adding them into the location. For example, imagine a questionnaire of 25 questions to each of which the respondent answers either 'YES' or 'NO' (True or False, 1 or 0). We could then pack the responses for each person into one memory location (as long as our memory word is at least 32 bits long, as it will be on any mainframe). These responses can then be compared, using logical operators such as .AND. and .OR. If two responses are stored packed into locations A and B, then A.AND.B will tell us all the questions on which person A and person B both answered 'YES' (if we unpack the result); A.OR.B will tell us on which questions either one or both of them answered 'YES'; and so on.

The system you are on may not let you store the results of numeric operations into a LOGICAL location, and it is quite likely that it will not let you perform logical operations such as .AND. and .OR. on integer variables. Yet we need to do integer calculations to pack the values, and then perform logical operations on the result. We can, however, "fool" the computer to allow us to do this job by using EQUIVALENCE; that way we can alternately treat the same location as integer and as logical. We will write out the program segment

which will read in the responses for person A, and pack them into a location that we can manipulate and compare using logical operations.

```
***** READ IN RESPONSES FOR ONE PERSON AND PACK *****
CHARACTER ANSWER(25)
INTEGER PACK
LOGICAL A
EQUIVALENCE (PACK, A)
READ 5, ANSWER
5 FORMAT(25A1)
MULT = 1
PACK = 0
DO 20 I = 1, 25
  IF(ANSWER(I).EQ.'Y') PACK = PACK + MULT
  MULT = MULT*2
20 CONTINUE
{packed "logical" value A can now be compared with B}
```

We assumed the answers to the questionnaire were single-letter responses, 'Y' or 'N'. A negative response was a zero in our binary true/false notation, and so did not have to be added in. Rather than taking each positive response and adding it to PACK by adding in  $2^{**}(I-1)$ , we saved time and computational expense by simply having MULT represent, as the loop progressed, the appropriate power of 2 that should be added in.

We will see in Chapter 13 that sets can also be represented and manipulated by this binary packing procedure.



## LARGE INTEGERS

We have seen that there are limits to the size of integers we can store in one memory location, depending on the word size of our machine. On a 32-bit machine, the largest integer value we can store is 2147483647 (that is,  $2^{31} - 1$ ), on a 64-bit supercomputer the largest integer value we can store in one memory word is 9223372036854775807 (that is,  $2^{63} - 1$ ). Yet there might arise occasions in which we need to do calculations with bigger integers than these (large prime numbers, factorials, and so forth). Analogous to the machine's automatic way of providing more significant digits precision for reals when we need them by making available DOUBLE PRECISION (that is, two locations to store the information rather than one), we can use a similar approach, and store our large integers in an *array* of locations.

The simplest (but least efficient) way to handle a large integer would be to store one decimal digit per array location. If we were reading in a large (say, 50-digit) integer into an array LARGE, dimensioned to 50 locations, we could do it by:

```

INTEGER LARGE(50)
READ 6, LARGE
6 FORMAT(50I1)

```

However, if our memory words will accommodate, say, 10-digit integer values, then we could fit the value into a much smaller array by reading 10 digits into each location:

```

INTEGER MAGNA(5)
READ 7, MAGNA
7 FORMAT(5I10)

```

These arrays containing large integers could then be added, subtracted, multiplied, and so forth, a block at a time, beginning at the right (end) of the arrays. Information such as a "carry" or a "borrow" would have to be carried over from one block to the block to its left (that is, above it in the array).

Let us try this technique by creating a program that will read in two large integers A and B in 3 two-digit "blocks" each. We have chosen a small block size and a short array to make the process *visible* using actual values. Notice, however, that once the program is checked out it will work just as easily on much larger arrays of bigger block sizes (we can choose the block sizes to approximate the integer capacity of one word in memory on our machine). In our example, we will have our program multiply 112233 (A) by 445566 (B). The maximum value of each block is 99 ( $10^2 - 1$ ); any results that exceed this value must be carried over to the next block. Let us first visualize the problem by laying out the operation as we would do it by hand, if we were multiplying together pairs of 2-digit blocks at a time.

(A)		11	22	33			
(B)		44	55	66			
(carry)		(	7	14	21	)	
(product)		(		26	52	78	)
(prod1)*			7	40	73	78	
(carry)		(	6	12	18	)	
(product)		(		05	10	15	)
(prod2)*			6	17	28	15	
(carry)		(	4	9	14	)	
(product)		(		84	68	52	)
(prod3)*			4	93	82	52	
(C)		5	00	07	20	88	78

From this pencil-and-paper exercise, we learn a few things about implementing our multiplication algorithm. At each stage, we take a block (2-digit integer) from B, and successively multiply it by each of the blocks in A, from right to left. The product of each block-pair may exceed 99, in which case we have a carry to the next block to the left. This carry must be added into the product of the next block-pair, and then we must test the result against 99 to see if there is a carry to the next block. After doing this against all of the blocks in A, we must proceed to the next block in B, and multiply it by all of the blocks in A, from right to left. This gives us our next partial product, also in block form. This repeats until we have used all of the blocks in our multiplier B. Then all of the partial products must be added up, a block at a time, from right to left, taking carries into account.

One way to implement this would be a two-dimensional array to hold the blocks of the respective partial products. We could then add them all up when the multiplication was finished. But we should be able to bypass this by just using one single product array C, and another array for the carries. C will then keep track of each block result, adding in new products and keeping track of carries as necessary. Each new product of a block-pair gets added into the appropriate block of C, and the result then tested for a carry. The size of C should be the sizes of A and B combined, since the product of two 6-digit integers may be as large as 12 digits (or the product of two 3-block numbers may be as long as 6 blocks; etc.). C and the CARRY array must be initialized to zero.

```
*****
*
*      PROGRAM TO MULTIPLY LARGE INTEGERS
*
*****
INTEGER A(3), B(3), C(6), CARRY(6), PROD
DATA C, CARRY / 12*0 /
READ 5, A, B
5 FORMAT(3I2)
DO 20 I = 3, 1, -1
    DO 10 J = 3, 1, -1
        K = I + J
        PROD = B(I)*A(J) + CARRY(K) + C(K)
        CARRY(K) = 0
        IF (PROD.GT.99) THEN
            CARRY(K-1) = PROD/100
            PROD = MOD(PROD, 100)
        ENDIF
        C(K) = PROD
10     CONTINUE
10
```

```

        C(K-1) = CARRY(K-1)
        CARRY(K-1) = 0
20  CONTINUE
*
          PRINT OUT PRODUCT
*
          DETERMINE FIRST NON-ZERO ENTRY
          L = 1
25  IF ( C(L).EQ.0 .AND. L.LT.6) THEN
    L = L + 1
    GO TO 25
ENDIF
PRINT 33, ( C(I), I = L, 6)
33  FORMAT(5X, 'THE PRODUCT IS ', 6I2)
STOP
END

```

Unfortunately, leading zeros in the inner blocks will not print in this program, but we will see a way to fix that in the next chapter. (See Chapter 11, the section on other format descriptors.)

One can readily see how to modify this program to handle larger integers, and use larger blocks. The block size will be determined by the largest integer that can be stored on your machine. Since two blocks are multiplied together to get a block product, the block size must be roughly half the largest integer size on the machine. On a 32-bit machine, we would probably choose a 4-digit block size to be safe. The length of arrays A and B will then be determined by the length of the integer to be multiplied divided by the block size. C should be of a length equal to the sum of the lengths of A and B.

If we assume a 4-digit block size, an A value 20 digits long and a B value 15 digits long, we would make the following changes in the preceding program:

```

INTEGER A(5), B(4), C(9), CARRY(9), PROD
DATA C, CARRY / 18*0 /
...
5  FORMAT(5I4)
      { and B must be entered right-adjusted in the first 16 columns}
DO 20 I = 4, 1, -1
    DO 10 J = 5, 1, -1
    ...
        IF (PROD. GT. 9999) THEN
            CARRY(K-1) = PROD/10000
            PROD = MOD ( PROD, 10000 )
        ...
25  IF (C(L) .EQ. 0 .AND. L .LT. 9) THEN
    ...
    PRINT 33, (C(I), I = L, 9)
33  FORMAT(5X, 'THE PRODUCT IS ', 9I4)

```

You can readily see how to modify the program for other block sizes and integer lengths.

This method of handling large integers can be extended to do addition, subtraction, compute large factorials, and so forth. You could even modify it to handle real values to a greater precision than DOUBLE PRECISION allows on your machine!



## A SUBTLE WAY TO INTERCHANGE VALUES, USING EQUIVALENCE

If numeric (integer or real) or character values (up to the length that is stored in one word in memory on the system) are equivalenced to logical variables (X and Y, say), the following operation will *interchange* the contents of X and Y:

```
LOGICAL X, Y
...
X = X .XOR. Y
Y = X .XOR. Y
X = X .XOR. Y
```

The .XOR. operator is a logical operator to take the *exclusive or* of two values, available on some systems, such that it is true if *either* of the two arguments is true (but not both), and it is false otherwise. If it is not available on your system, you can write a statement function which will do the job:

```
LOGICAL XOR, X, Y
XOR (X, Y) = X .OR. Y .AND. .NOT. (X .AND. Y)
...
X = XOR (X, Y)      etc.
```

or the Standard logical operator .NEQV. does the same job. The credit for this idea goes to Professor Charles Redeker.

Thus, if you EQUIVALENCE integer variables (M and N), or real variables (A and B), or character variables (C and D) to the logical variables (X and Y), the logical manipulations indicated will interchange the values of the variables involved.

```
LOGICAL X, Y
REAL A, B
INTEGER M, N
CHARACTER C, D
EQUIVALENCE (X, A, M, C), (Y, B, N, D)
A = 2.5
```

```

B = 3.75
X = X .XOR. Y
Y = X .XOR. Y
X = X .XOR. Y
***** CONTENTS OF A AND B HAVE BEEN INTERCHANGED *****
PRINT*, A, B
M = etc.

```

Of course, this will also work to interchange the contents of two *logical* variables. The limitation of character variables that can be interchanged is dependent of the way character variables are stored on your system. On *any* system, character variables one character long can be interchanged this way; beyond that, character variables of the length which is stored in one word in memory on your system can be interchanged using this method (for example, on a VAX/VMS system, which has 32-bit words and 8-bit bytes, character variables are packed 4 characters per word in memory, so variables up to four characters long can be switched).

This idea can be expanded to write a subroutine which will interchange any two integer values, or two real values, or two logical values; it will even interchange two character values (within the length limits discussed) if they are equivalenced to two logical variables and those variables are passed.

```

***** DEMONSTRATION OF GENERAL-PURPOSE SWITCH ROUTINE *****
CHARACTER CAT, DOG
LOGICAL X, Y
EQUIVALENCE (X, CAT), (Y, DOG)
REAL A, B
INTEGER M, N
DATA A, B / 2.6, 8.7 / M, N / 5, 8/
X, Y / .TRUE., .FALSE. /
PRINT*, A, B
CALL SWITCH (A, B)
PRINT*, A, B
PRINT*, M, N
CALL SWITCH (M, N)
PRINT*, M, N
PRINT*, X, Y
CALL SWITCH (X, Y)
PRINT*, X, Y
CAT = 'C'
DOG = 'D'
PRINT*, CAT, DOG
CALL SWITCH (X, Y)
PRINT*, CAT, DOG
STOP
END

```

```
SUBROUTINE SWITCH (X, Y)
LOGICAL X, Y
X = X .NEQV. Y
Y = X .NEQV. Y
X = X .NEQV. Y
RETURN
END
```

If your system balks at passing arguments that are integer or real to a subroutine whose dummy arguments are logical, you can use the same trick for the integer and real values that we used for the character variables—equivalence them to logical variables in the calling program, and pass the logical variables to the subroutine. This gives you a *very* general-purpose interchange routine that would not be available otherwise.

## FORTRAN 90 FEATURES

A feature found in several languages today is the “case” structure, in which different segments of code may be executed if a particular expression matches the value or range of values specified for that case. Fortran 90 will implement a SELECT CASE feature of the form:

```
SELECT CASE (case-expression)
CASE case-selector1
    action1
CASE case-selector2
    action2
...
[ CASE (DEFAULT) ]
END SELECT
```

but until that is available, case conditions can be expressed as the logical conditions in IF/THEN/ELSEIF structures to accomplish the same job.

## SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

A way of testing a variety of values and taking action based on the appropriate value (besides the “case” structure) is the Computed GO TO. This control structure has the form:

GO TO ( $n_1, n_2, \dots, n_m$ ) [,] intexp

where the integer expression (intexp) should evaluate to a value between 1 and m (the number of statement numbers in the list following the GO TO); if intexp is 1, a branch to the first statement number in the list ( $n_1$ ) is taken, if intexp is 2, a branch to  $n_2$  is taken, and so on. If intexp is not in the range from 1 through m, the statement is skipped, and the next executable statement following the computed GO TO is performed.

The ASSIGN statement (not much used) assigns a statement number (n) to an integer variable (i):

ASSIGN n to i

so that i may then be used as a target in a GO TO statement:

GO TO i [( $s_1$  [ $s_2$ , ...])]

which may optionally be followed by a set of statement numbers (s's) which define the range of allowable branch statements.

The EQUIVALENCE statement may be used to "patch" typographic errors, to equate large areas of memory to save space or to make handling of multi-dimensional arrays more efficient, or to treat a location alternately as different types of variables. This latter feature allows for some elegant manipulation of values, as shown in the text. The form of the EQUIVALENCE statement is:

EQUIVALENCE (var<sub>1</sub>, var<sub>2</sub> [...]) [, (var<sub>n</sub>, var<sub>k</sub>, ...)]

where the variable names which are equated may be of different types, or may be beginning locations of arrays.

There are restrictions on the use of EQUIVALENCE with COMMON; you cannot EQUIVALENCE two variables which are both in COMMON, and if you EQUIVALENCE a variable in COMMON to one not in COMMON, it must not extend the COMMON block upward.

Small integer values or logical values (1,0) can be "packed" several to one storage location, by multiplying by powers of ten or two, and adding up the scaled values. They can subsequently be "unpacked" using MOD and integer truncation. Such savings of storage space may be necessary in certain circumstances.

Some large integer values may exceed the word capacity of a computer location. To handle such large values, they can be broken up into "chunks" and stored in locations in an array. The appropriate manipulations (addition, subtraction, etc.) can then be done to the array elements, taking carries into account, and the results printed out or stored on disk.



## EXERCISES

1. The following algorithm purports to determine the day of the week on which any given date (from 1600 to 2000) falls (or fell). Write a FORTRAN program (you may find using ELSEIF constructs and arrays useful) to imple-

ment the algorithm, and print out the day of the week on which any given input date falls. Note that you must express the year in its full 4-digit form. Decide whether you want to input dates in a form such as December 7, 1941 or 12/7/1941, and then set up your input formats accordingly. Try it out for at least five different dates.

Pick any date, 1600 to 2000. Add 1/4 (truncated) of the last two digits of the year to the last two digits of the year. Add a value depending on what month your date falls in:

MONTH	ADD
January	1 (for a leap year, add 0)
February	4 (for a leap year, add 3)
March	4
April	0
May	2
June	5
July	0
August	3
September	6
October	1
November	4
December	6

Add the day (that is, the date in the month). Now add a value according to the year:

1900	-	2000	Add 0
1800	-	1899	Add 2
9/14/1752	-	1799	Add 4
1700	-	9/13/1752	Add 1
1600	-	1699	Add 2

Calculate the remainder when your total is divided by 7. The remainder represents the day of the week on which your date falls such that 1 is Sunday, 2 is Monday, . . . , 0 is Saturday.

Because of the range of possible dates you may use, you need to use the more detailed leap year test. A year divisible by 4 is a leap year, *except* that years divisible by 100 (century years) are only leap if they are also divisible by 400.

- ◆ 2. The algorithm in problem 1 works very nicely for this century (the 20th Century), and could be used to make your calendar program (exercise 19 at the end of Chapter 7) more elegant, so that the user only had to input the year for the calendar, and the program would determine the day of the week on

which the first of January falls. Notice that the algorithm in problem 1 can be greatly simplified for this, since the date will always be the same—January 1, and the year will be in this century.

3. Write a subroutine (a skeleton beginning follows) that will take an integer argument N (that lies between 1 and 365; ignore leap years) that represents a day in the year. The subroutine must translate the integer into the actual date it represents (for example, an N of 41 would represent February 10). After the subroutine has determined the date, it must also print out the astrological sign for someone born on that date. The signs of the Zodiac *begin* their influence on the following dates:

Aquarius	January 20
Pisces	February 19
Aries	March 21
Taurus	April 20
Gemini	May 21
Cancer	June 21
Leo	July 23
Virgo	August 23
Libra	September 23
Scorpio	October 23
Sagittarius	November 22
Capricorn	December 22.

The arrays provided in the subroutine skeleton may be helpful.

```

SUBROUTINE DATES (N)
CHARACTER SIGN(12)*11, NAME(12)*10
INTEGER MONTH(12)
DATA SIGN / 'AQUARIUS', 'PISCES', 'ARIES', . . .
DATA NAME / 'JANUARY', 'FEBRUARY', 'MARCH', . . .
DATA MONTH / 31,28,31,30,31,30,31,31,30,31,30,31 /

```

The program output should be something like:

```
DAY # 41 IS FEBRUARY 10, AND THE SIGN IS AQUARIUS
```

4. In the game of craps, a 7 or an 11 on the first roll is a win, and a 2, 3, or 12 is an instant loss. Any other roll means that you have to play for a "match," that is, try to match what you rolled (a win) before you roll a 7 (a loss). Assume that a subroutine exists which will "roll" the dice for you, and write a program which will play 100 complete games of craps, and count how many are won and lost. The subroutine will simply give you a roll value from 2 through 12.

Use a computed GO TO to branch to the appropriate statement in the program given the roll.

- 5. Write a program which will fill a two-dimensional array with values (such as defining  $A(i,j)$  as  $i^2 + j^2$ , or  $1.0/(i+j-1)$  [the Hilbert matrix], where  $i$  and  $j$  are the subscripts). EQUIVALENCE the two-dimensional array to a one-dimensional array of the same size, and use the one-dimensional array to calculate the average value in the two-dimensional array you filled.
- 6. Set up a character array that will contain a letter you intend to use when you begin looking for a job. Write your program so that it is very easy for you to change the inner address of the person to whom it is addressed, his or her name in the salutation, and any references to the company name that occur in the body of the letter.
- 7. Now that you have arrays available, modify the histogram example at the end of Chapter 6 so that it will print the histogram at the *left* of the output page.
- 8. Write a program which will read in a character string representing a *real* value (assume it contains a decimal point, has no exponential [E] notation, and fills 10 columns), and convert it to its equivalent real numerical value. You may want to use the CHAR/ICHAR system functions.
- 9. Using the CHAR/ICHAR functions and arrays, write a program that will read in text in lowercase letters, convert all the lowercase letters to capitals, and output the result. Make your program sophisticated enough that it only changes the letters of the alphabet, and leaves other symbols (such as ',') alone.
- 10. Write a program which will *unpack* the 20,000 age values that were packed in the PACKUP program in this chapter, in the order in which they were packed, determine the largest and smallest values in the collection, and count each value from 0 to 21.
- 11. Write a program which will accept two large integers (say up to 40 digits long) and a command to add or subtract them. Perform the indicated operation, and print out the input values and the result, in a meaningful format.
- 12. Write a program which will determine and print out, to any specified number of significant digits, the result of dividing one integer value by another (for example,  $1/7$ ).
- 13. Use the technique developed in problem 11 to determine and print out, to 100 significant digits, an approximation for pi from the following formula:

$$\pi = 4 ( 1 - 1/3 + 1/5 - 1/7 + 1/9 - \dots )$$

Either add 1000 terms, or else use an epsilon test to keep adding terms until the new approximation differs from the last by less than  $1.0 \times 10^{-100}$ .

- 14.** Use the technique of handling large integers to print out a table of factorials up to 75! (which has 110 digits).
- 15.** Write a subroutine (or a function) which will accept N scores ( $N \leq 15$ ), from judges for an Olympics gymnastics event, throw out the high and the low score, and print out the average of the rest. (This technique could also be used to average measurements taken in your laboratory, where you might consider the highest and lowest measurements as possibly erroneous.)
  - 16.** A useful subroutine could be designed to perform money exchange rates for many of the major countries. It should include an array of the names of the countries, a second array which indicates the currency denomination, and a third array which has the current exchange rate. Then a country name and a dollar and cents amount (in U.S. currency) could be passed to the subroutine, which would print out the equivalent amount in the currency of that country. The subroutine would accept a country name (character string, say 26 characters) and a real amount to represent the U.S. currency amount; it would then print out, for example, \$40.26 = 23.99 pounds. Data for the arrays could be derived from a table such as this (5/24/91 exchange rates):

Argentina/austral	.000101	France/franc	.1720
Australia/dollar	.7575	Greece/drachma	.005352
Austria/schilling	.0831	India/rupee	.0480
Belgium/franc	.0284	Israel/shekel	.4237
Brazil/cruzeiro	.0071	Italy/lira	.000788
Britain/pound	1.7315	Japan/yen	.007226
Canada/dollar	.8703	Kuwait/dinar(na)	(3.5 in 1988)
Colombia/peso	.00167	Mexico/peso	.000333
Denmark/krone	.1526	Netherlands/guilder	.5184
Egypt/pound	.3089	S. Africa/rand	.3587
Finland/mark	.2494	Germany/mark	.5869

The amounts given in the table are U.S. dollar equivalents.

- 17.** Write a subroutine which will accept a character (or integer, if you prefer that notation) two-dimensional array of any size, which is filled with blanks (or zeroes) and some character (or integer) representing a two-dimensional physical object. Have your subroutine calculate and print out the coordinates (in the array) of the *center of gravity* of the object. You may assume that each "square" in the array is weighted equally (unless you want to get fancier, and have the integer values in an integer array represent different weights of that segment of the whole body). A weighted sum of area weights times the (x- or y-coordinates), divided by the summed weights, will give the appropriate center-of-gravity coordinates.

- 18.** Another method for determining the day of the week for a date with month number M (where M is 1 for March, 2 for April, etc., 11 for January and 12 for February), D is the day of the month, C is the century (first two digits of the year), and Y is the year within the century (last two digits of the year) is called Zeller's congruence, after the Reverend Zeller:

$$W = \{ [2.6M - 0.2] + D + Y + [Y/4] + [C/4] - 2C \} \bmod 7$$

where [ ] represents the truncation (integer part) operation, and W will be a value from 0 to 6, where 0 represents Sunday, 1 represents Monday, and so on. Write a program (a subroutine would seem appropriate) which will accept a date in terms of month, day, and four-digit year, and compute and print out what day of the week it falls on, according to Zeller.

- 19.** You may well have created a reference array in problem 18 containing the names of the days of the week, such as:

```
CHARACTER*9 WEEK(7) { or WEEK(0:6) }
DATA WEEK/ 'SUNDAY', 'MONDAY', 'TUESDAY', . . . /
```

Make use of this array and others you have used in "date" problems to write a subroutine to solve the following problem: you have been given a year and a number (1 – 366) which represents the day number in the year of a particular date. Convert the day number into a date in terms of month and day (for example, day number 45 is February 14). Assume that you are also told the day of the week on which January 1 falls for that year, and use this information to determine the day of the week for your date. The output should be something like: February 14 was a Tuesday.

- **20.** Using your date-problem arrays, determine the number of days between any two dates in the same year. The year will have to be specified, so you know whether it is leap. The two months may be the same. Print out the dates and the number of days that lie between the two dates.

- 21.** Write a subroutine which will read in a large integer value and output the value with commas inserted properly every three digits, working from the right. For example, an input value of 2345678901 should output 2,345,678,901.

- 22.** *Cryptography.* Much work is done in coding and decoding messages these days. A simple code is to map the original letter of a message onto a different symbol according to some pattern (a "substitution cipher"). A simple substitution pattern can be gotten just by reversing the order of the letters of the alphabet (so that 'A' codes into 'Z', 'B' codes into 'Y', 'C' codes into 'X', and so on). Using the CHAR/ICHAR functions, create two subroutines, one which will read in a message in capital letters and *encode* it according to this mapping, and the second to *decode* a coded message and print out the deciphered result.

Modify the two subprograms you have written to use a different pattern, which makes the coded letters a *shift* of the original alphabet ordering; for example, a shift of 5 places is:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
V W X Y Z A B C D E F G H I J K L M N O P Q R S T U
```

You *could* create parallel arrays, but with a little thought, you can probably find a more clever way to do the transformations.

- 23.** A telephone has the following letter equivalences for the digits on the dial:

ABC = 2	MNO = 6
DEF = 3	PRS = 7
GHI = 4	TUV = 8
JKL = 5	WXY = 9
(possibly QZ = 0)	

let us assume the 1 then represents a blank. Read in a 7-digit phone number and print out all of the "words" on the dial that could be used to represent that phone number.

- **24.** A "sparse" large matrix (two-dimensional) may be stored more efficiently by not taking up the memory space for the entire matrix, mostly filled with zeros, but rather using another array to store only those nonzero values and their locations. You will also need a scalar value to represent the number of nonzero values that are stored. Write a program which will store this information in a sparse array. Then write code to add together two such sparse arrays and store the result. Write code to output the contents of such stored information (including the implicit zeros) to an output device as a full  $n \times m$  array.
- 25.** The *two's complement* of a binary value is the *one's complement* plus 1. Refer to Appendix A for more information on binary representations and one's complements. Now use your knowledge of EQUIVALENCE and LOGICALs to take an integer value and create its two's complement; print out the result.
- 26.** Read in a text several pages long and calculate the average word length in the text. (Information such as this is used in trying to determine authorship of a newly discovered work.)
- 27.** Write a function which will convert a Roman numeral into an integer value. Write a second function which will convert a positive integer value (less than 4000) to a Roman numeral string. (Warning: these are not trivial

exercises.) The Roman numeral equivalences are:

$$\begin{array}{ll} 1 = I & 50 = L \\ 5 = V & 100 = C \\ 10 = X & 500 = D \\ & 1000 = M \end{array}$$

There can be no more than three repetitions of a symbol in a row, so we count: I, II, III, IV, V, and so on. The symbols for the first three powers of ten (I, X, C) can be subtracted from symbols up to and including the next power of ten to create values (such as IV, IX, XL, XC, CD, CM) that avoid repetitions of four symbols in a row.

- **28.** A “check” digit can be used to determine if data has gone “bad.” If you have a file of measurements expressed in integers up to 9 places, add a 10th digit to make the sum of the digits odd.

# CHAPTER 11



## FILE HANDLING AND OTHER ADVANCED INPUT/OUTPUT

In order to handle the large quantities of data which are to be manipulated by some scientific programs, we need to use mass storage media such as disks or tapes. The special rules that govern file handling on such devices should be part of your repertoire. There are additional special features of output design which can make the results of your programs more eye-catching and informative.

The hard disk drive of a computer allows the user to access files efficiently.

*"Open Sesame!"*

- "The History of Ali Baba," Arabian Nights

For much of input/output, we found that list-directed statements were adequate. When we wanted more control of the I/O, we used simple FORMAT statements. These FORMATS included carriage controls (for printed output), spacing (X), and simple format descriptors for characters (Aw), integers (Iw), reals (Fw.d, Ew.d, Gw.d), and double precision (Dw.d). In this chapter, we will increase your collection of I/O tools.

## ◆ OTHER FORMAT DESCRIPTORS

### Integer Output Revisited

The integer format descriptor, usually Iw, where w indicates the field width, can be more finely defined to indicate how many significant digits you want to see in the integer output field. The format descriptor Iw.m defines a field width w in which *at least* m significant digits will be output. The value of m must not be larger than w.

This convention can be used to force the printing of leading zeros in cases where they would be desirable. For example, it could have been used in our large-integer multiplication problem of the last chapter to print out all the digits (including leading 0s) of all blocks after the first. In the case where the result might be 9 blocks long, the format statement to output the array of values could be modified to read:

```
33 FORMAT(5X, 'THE PRODUCT IS ', I4, 8I4.4)
```

### Numeric Signs

As you are aware, on normal output a negative value will be preceded by a minus sign (-), but a positive value simply has no sign. If you want to force output of plus signs ('+'s) for positive values, include an SP edit specification in your FORMAT statement; this will force the output of a sign for all following numeric items in the FORMAT statement. An SS edit descriptor in the FORMAT will "turn off" the effect of the SP in the editing.

## Scaling

The edit control nP can be used to *scale* values on input or output. Normally a FORMAT has a scale factor of zero, which remains in effect throughout. If you include a scale factor P in the FORMAT statement, it will remain in effect for the rest of the FORMAT, unless changed by another scale factor; to set the rest of the FORMAT back to normal, put in an OP edit scale factor.

On input, a scale factor kP for a value input with an F, E, D, or G format with no exponent in the field, the value in the field is divided by  $10^k$  before it is stored. If there *is* an exponent in the field, the magnitude of the value stored is unchanged from that of the value in the input field.

On output, a scale factor kP applied to a real value output with F format will *multiply* the internal value by  $10^k$  before it is output. If E or D format (or G format that converts to E) is used, the real multiplier (mantissa) that would be output is multiplied by  $10^k$  for the output display, and the exponent is reduced by k. In this latter case, the *value* output is not different from the internal value, but the way in which it is displayed on output is changed.

Some examples to illustrate how P scaling works (and cases in which it is ignored) are shown in the following tables:

Input Value	Input Format	Stored Value
-15.678	2P,F7.3	-.15678
99.9999	-2P,F7.3	9999.99
123.45	3P,E8.2	.12345
6.1E 23	4P,E8.2	.61E+24 {no effect}

Stored Value	Output Format	OutputValue
987.65	1P,F7.2	9876.50
.334 E 2	-1P,E10.4	.0334E+03
.987 E 5	3P,E9.1	987.0E+02
710.8	-2P,F6.3	7.108

Note that, for real values using F format, in both input and output, the *external* value (the value either in the input record read in, or the value output to the output device) is  $10^k$  times the *internal* value (the value in computer memory).

The scaling editor could be used, for example, to convert input values expressed in centimeters to meters for the internal storage values by using 2P before the input format descriptor. Similarly, values which are calculated internally in terms of grams may be displayed as kilograms on output by using -3P.

## BN and BZ Editing, Discussed Further

By default, your system has a convention regarding how trailing blanks in numeric input fields are to be interpreted. On some systems they are ignored, and on others they are interpreted as zeros. This makes a big difference when the following input record is read with an I6 format:

35	{I's mark the edges of the field}
----	-----------------------------------

If blanks are ignored, the value stored will be 35, but if blanks are interpreted as zeroes, it will be stored as 3500. Try this out as a quick exercise on your computer, to determine what the system default is (or look it up in the system manuals; but we predict the “check-it-yourself” test is easier and quicker).

You can control how blanks are to be interpreted yourself by inserting one of the following edit controls in your FORMAT:

BZ – blanks are interpreted as zeros

BN – blanks are ignored

From the point in the FORMAT at which one of these is encountered, it determines how blanks will be interpreted for input values for the rest of the statement (unless another B control is met).

Thus, for the following input FORMATS:

```
33 FORMAT(I8, BZ, 3I6, I5)
44 FORMAT(BZ, 3I4, 2X, I5, BN, 5I3)
```

in FORMAT 33, the first field (I8) will have blanks interpreted according to the system convention, and the rest of the fields (3I6, I5) will have blanks interpreted as zeros; in FORMAT 44, the first formats (3I4, I5) will have blanks interpreted as zeros, but the remainder (5I3) will have blanks ignored.

The blank convention can be set for a particular external file by setting a parameter in the OPEN statement (as we shall see in a few pages). Even this convention can be overridden on READs from such a file by using BZ or BN in the READ formats.



## TABBING (T FORMAT)

You may be familiar with the Tab key on a typewriter or a computer keyboard, which allows you to set spacing to definite columns of the output page. The T edit descriptors in FORTRAN allow you to perform similar controlled motions in I/O. The edit descriptor Tn moves to position n of the I/O field, and whatever format descriptor appears next in the FORMAT statement is applied beginning in that column. For example, in the statements:

```
READ 44, KING, KONG
44  FORMAT(T5, I4, T12, I5)
```

the READ operation will “tab” to column 5 of the input record and then read an I4 field (that is, from columns 5–8); the value will be stored in the integer variable KING. Then the READ will “tab” to column 12 of the record, and read from a five-column field (that is, from columns 12–16), into integer variable KONG.

On output, the tabbing operation is slightly more complex, since Tn tabs to position n of the output record being created by the FORMAT statement. This output record includes, for screen and printed output, the *carriage control* character in the first position, a character which is never displayed in the visible output but merely controls vertical spacing. For the visual output displayed, Tn results in a format specification beginning in output column n – 1. Thus, in the following program segment:

```
A = 3.4
MAN = 123
PRINT 55, A, MAN
55  FORMAT('0',T10, F3.1, T3, I3)
```

the PRINT record will have a ‘0’ in position 1 (for carriage control to skip a line), it will “tab” to position 10 and put A in positions 10–12 (which will be output columns 9–11), and then it will tab *back* to position 3 (output column 2) and display the integer value of MAN (123) in the three columns beginning there.

{skips a line}  
123 3.4

### Relative Tabbing

The tab edit controls TRn and TLm will tab right or left, respectively, relative to the last position indicated in the format record. Notice carefully that the Tj descriptor is an *absolute* position reference, indicating that the next format

operation will occur beginning at character position *j*; but the TRn and TLM controls tab *relative* to the last position specified. Thus, in the following output statements,

```
PRINT 66, 'L', 'I', 'N', 'E', 'A'  
66 FORMAT(T4, A1, T5, A1, TR2, A1, TL1, A1, T3, A1)
```

the word ALIEN will be output beginning in output column 2 of the screen or printer. Since no carriage control was explicitly specified, it was a blank, and thus printed on the next line down.



## DIRECT AND SEQUENTIAL ACCESS FILES

Up to this point, our only input has been from the keyboard (though on an older batch system we might have had to use some punched cards on which our input data was recorded), and our only output has been to the screen and/or the printer. Occasionally you will have to deal with data in larger quantities, or with information that needs a more permanent life than that we have used so far. In such cases, you can make use of *external files* which contain data to be input to your program, or which you may WRITE as output from your program. These external files are usually on magnetic disk or tape, and are either direct or sequential access.

A *sequential access* file is one in which the information must be read (accessed) in the order in which it was written out. It is analogous to a cassette tape on which you have ten songs recorded; if you want to hear the fifth song, you have to "fast forward" past the first four songs to get to it. Computer devices that are sequential access include the card reader and magnetic tape drives. To read the 100th card in a punched-card deck, you first have to flip the 99 cards in front of it through the card reader; to read the 50th record on a tape (which, when mounted on the tape drive, was positioned at the beginning, "load point") you have to read past the first 49 to get to it.

A *direct access* file is one in which any piece of data can be reached in roughly the same time as any other piece of data. It is analogous to the same set of ten songs mentioned earlier, but this time recorded on a phonograph record; you can drop the needle precisely on the band with the song you want to hear (or to a compact disk [CD], in which your CD player allows you to key in which track you want to hear). The most important direct access device for our computer use is the disk drive. On a microcomputer, you may use "floppy disks" or have a "hard disk" installed; on a mainframe data may be saved on many disks (like platters, analogous to our phonograph records) that may be mounted, up to 11 at a time, parallel to one another in a disk pack. You may have special information stored on a disk and have to ask to have it mounted

in a drive for your program, or the system may have permanent disk space directly accessible by your program on which you can write the information you need.

Information written out to disk may be either "sequential" or "direct," but that written out to tape must be "sequential." A number of special commands are needed to access these external files for use by your program, and we will discuss these next.

## OPEN Clause

To "connect" the file you want to use to your program, you must execute an OPEN statement, which minimally must specify the unit number of your file and the file name; it also should include its status ('NEW' or 'OLD'). Thus the following OPEN statement:

```
OPEN (4, FILE = 'TEMPS.DAT', STATUS = 'NEW')
```

would connect unit 4 to your program to create a new file called 'TEMPS.DAT'. Your program could then execute WRITES to unit 4 that would output data to this file.

There are several other specifiers (besides the unit number, file name, and status indicator) that may optionally be used in the OPEN statement. A complete list of these specifiers follows.

**UNIT=** (or **UNIT =**) [optional] You must always specify a unit number to be used for your file I/O. If this is the first piece of information in the OPEN statement, you may omit the phrase UNIT= and simply write the unit number. However, if you do not specify the unit number first, then you must use the identifier UNIT=. Thus, the following two OPEN statements are equivalent:

```
OPEN (4, FILE = 'MYFIRST', STATUS = 'OLD')
OPEN ( FILE = 'MYFIRST', STATUS = 'OLD', UNIT = 4)
```

**FILE=** You must specify a file name for any file that is to be saved after the end of the program. If you are just creating a SCRATCH file to contain temporary data during this program's execution, that will be deleted afterward, you do not have to bother with a name. The file name must conform to the naming conventions for files on your system (this may include certain "extensions" to the name, such as the .DAT we used earlier to indicate a data file), and it must be enclosed in single quotes.

**STATUS=** The STATUS of the file should be indicated, as a character string. This may be either 'NEW', 'OLD', 'SCRATCH', or 'UNKNOWN' (note that

these descriptors must be enclosed in single quotes). If 'NEW' is specified, there must not be an already existing file with the indicated file name; this OPEN statement will *create* a new file with that name. IF 'OLD' is indicated, there must be an existing file accessible by your program which has that name. 'SCRATCH' indicates that this file will just be used temporarily while this unit is connected to your program, and will be deleted when the unit connection is CLOSEd. Scratch files do not have to be given a name in your OPEN statement. The indicator 'UNKNOWN' makes the file status dependent on the default specified by your system (check your manual). If the STATUS specification is omitted altogether, the default value is 'UNKNOWN'.

**ACCESS=** If this specifier is included, it should be followed by one of two possible character strings: 'SEQUENTIAL' or 'DIRECT'. If the ACCESS= specifier is not used, the default is to *sequential* access. We will most often create sequential access files; if you do specify 'DIRECT' access, your OPEN statement must also include a *record length* specifier, RECL=.

**RECL=** This specifier must be followed by an integer, which indicates the record length (in terms of number of characters) of *each* record that will be written out to the file. RECL= is *only* used for direct access files, and must be omitted if your I/O is related to a sequential file. Notice carefully that *all* records written to a direct access file must be of the *same* length, the length specified in the RECL= clause; this restriction on record lengths does not apply to sequential files.

**FORM=** A file may be either formatted or unformatted. Most of the files we use will be formatted (that is, written or read using either a FORMAT designation or list-directed I/O). The default for this specifier is 'FORMATTED' if it is omitted. If you include the clause FORM = 'UNFORMATTED' in your OPEN statement, the I/O will be in terms of binary data, only readable by another computer of the same type. We will briefly discuss unformatted I/O in an upcoming section.

**BLANK=** This specifier, if used, must be followed by one of two character strings: 'NULL' or 'ZERO'. This indicates whether blanks in formatted numeric input fields will be ignored ('NULL') or treated as zeros ('ZERO'). If the specifier is omitted, the default is 'NULL' (that is, blanks are ignored). This is only used for *formatted* files. The convention specified by this clause can always be overridden by the appropriate BZ or BN edit descriptor in a FORMAT description, as discussed earlier.

**ERR=** If this clause is used, it should be followed by an existing statement number in the program; if an error occurs on OPENing the file (for example, if you are attempting to connect a file designated as 'OLD' which does not exist), your program will branch to the indicated statement number instead of abruptly terminating on the error condition. The next

specifier, IOSTAT=, can then be used by your program to determine what sort of error condition occurred, and take appropriate action.

IOSTAT= If this clause is included, it should be followed by the name of an integer variable (for example, IOSTAT = NERR). This integer variable will then be given a value indicative of what error has occurred in attempting to open the file. The value of the variable will be 0 if no error occurred, and it will have a system-dependent value if one does occur. You will have to check your system manual to see what values will be assigned for various OPEN errors. Once you know this, you can use the ERR= ns clause just discussed to keep the program alive if an error does occur, and at statement ns you can test the variable (for example, our NERR) to see which error condition has occurred, and then take appropriate action.

## ENDFILE Statement

When you are writing data to a sequential file, it is considered good form to put a special end-of-file (EOF) record at the end of the data you have written. Then when another program is reading the data from this file, it can recognize when it has encountered the end of the data, and thus stop reading (this is done using an END= clause, discussed in Chapter 6, in the section on input formats). To write an end-of-file record, you simply execute the ENDFILE command, specifying the unit number, in one of the following ways, where u is an integer representing the unit number:

```
ENDFILE (u)
ENDFILE (UNIT = u)
ENDFILE u
```

The ENDFILE statement may also optionally include IOSTAT= and ERR= clauses. It is not appropriate to write an end-of-file record at the end of data written out in *direct* mode, only that written in *sequential* mode.

## BACKSPACE and REWIND

To understand best how BACKSPACE and REWIND work, it is informative to visualize a collection of records written out on a reel of magnetic tape. The records are written out on this tape in sequential order, one right after the other (with a short, usually about 1/4", *interrecord gap*, between them, to allow for slight backup when the tape is stopped and restarted), beginning at the very beginning of the tape (the "load point"). After the records (and the end-of-file) have been written, if the tape is *rewound*, it will be positioned back at the

beginning (the load point). Thus a REWIND command goes back to the beginning of the very first sequential record written. The REWIND command must specify the unit number; optionally it may also include IOSTAT= and ERR= clauses (discussed previously). A REWIND command for unit u may take one of the following forms:

```
REWIND (u)
REWIND u
REWIND (UNIT = u)
or
REWIND(u, IOSTAT = i, ERR = j)
```

The BACKSPACE command, if we keep our magnetic tape picture still in mind, would “backspace” the tape to the beginning of the immediately previous record. Thus, we would be positioned at the beginning of the previous record, ready to READ or even WRITE (though this can be dangerous in sequential files). A BACKSPACE command must reference the unit number, and optionally may include IOSTAT= and ERR= clauses.

```
BACKSPACE (u)
BACKSPACE u
BACKSPACE (UNIT = u)
or
BACKSPACE (u, IOSTAT = ivar, ERR = j)
```

You may not BACKSPACE over records that have been written with a list-directed output statement.

## CLOSE

After you have finished the I/O with a file you have OPENed, you should CLOSE the file. This makes all the loose ends tidy before your program is completed; however, if you omit the CLOSE statement, at the end of your program the system will close all the files for you. The CLOSE command must at least specify the unit number (u) that is being closed:

```
CLOSE (u)      or      CLOSE u      or      CLOSE (UNIT = u)
```

It may also optionally include IOSTAT= and ERR= clauses, and a STATUS = *character string* clause which may specify either ‘KEEP’ or ‘DELETE’, indicating what the fate of the file should be. You cannot specify ‘KEEP’ for a ‘SCRATCH’ file. If this clause is not used, the default is ‘KEEP’, except for scratch files, which will be automatically deleted.

## INQUIRE

You may inquire about the properties of a file or a unit; this inquiry may be made whether or not the file/unit is connected at the time. The INQUIRE statement specifies either a unit (UNIT=u) or a file (FILE= character string), plus a set of "inquiry specifiers" which may include any of the following:

IOSTAT= ivar (integer variable ivar will have the value 0 if no error condition exists; otherwise it will take on a system-dependent value—consult your manual for the significance of the various values);

ERR= ns (where ns is a statement number that will be branched to if an error occurs on the INQUIRE);

EXIST= lex (where lex is the name of a logical variable which will be set to .TRUE. if (a) the file inquired after exists or (b) the unit specified exists; it will be .FALSE. otherwise);

OPENED= lex (where lex will be set to .TRUE. if (a) the file inquired after is connected to a unit, or (b) if the unit specified is connected to a file; it will be .FALSE. otherwise);

NUMBER= n (where, if the inquiry is about a specified file, n will be given the number of the unit it is connected to);

NAMED= lex (.TRUE. if the file has a name; otherwise .FALSE.);

NAME= c (where c is a character variable which will contain the name of the file, if it has one; otherwise undefined);

ACCESS= c (where c is a character variable that will be given the value 'SEQUENTIAL' or 'DIRECT' to indicate access mode);

SEQUENTIAL= c (where c is a character variable which will be set to 'YES' if sequential is included among the allowed access methods for this file; otherwise 'NO' or 'UNKNOWN');

DIRECT= c (where c is a character variable which will be set to 'YES' if direct is included among the allowed access methods for this file; otherwise it will be 'NO' or 'UNKNOWN');

FORM= c (where c is a character variable that will indicate the form of the data in the file—'FORMATTED' or 'UNFORMATTED');

FORMATTED= c (where c is a character variable that will be set to 'YES' if the file connected is formatted; it will be set to 'NO' or 'UNKNOWN', if it cannot be determined);

UNFORMATTED= c (where c will be set to 'YES' if the file is unformatted; otherwise set to 'NO' or 'UNKNOWN');

RECL= n (where n is an integer variable that will be set to the record length for a direct access file; otherwise undefined);

NEXTREC= n (where n is assigned the value k+1, if k is the record number of the last record read or written in a direct-access file; if it is not direct access, n is undefined);

BLANK= c (where c is a character variable that will be given the value 'NULL' if the blank control for the file is that blanks will be ignored, 'ZERO' if the control is set so that blanks are interpreted as zeros; if the file is not formatted, c is not defined).

Note that you will probably only want to use a few of these inquiry specifiers if you inquire about a file or a unit.

## A Simple Sequential-Access Program

To illustrate some of the file handling techniques we have just described, let us write a simple program that will create a sequential file of temperature measurements made at uniform intervals during a chemical reaction. We will characterize the temperatures as being entered at the terminal, but a more sophisticated (analog-to-digital) arrangement would allow them to be fed in directly from a temperature monitoring device to the program (similar arrangements could be used for medical data, etc.).

```

PROGRAM CHEMIS
REAL TEMP
INTEGER MEAS
PARAMETER (MEAS = 100)
OPEN (3, FILE = 'TEMPS.DAT', STATUS = 'NEW')
DO 30 I = 1, MEAS
    READ*, TEMP
    WRITE (3, *) TEMP
30 CONTINUE
ENDFILE (3)
CLOSE (3)
STOP
END

```

A program which would then read from this sequential file would have an OPEN statement:

```
OPEN (2, FILE = 'TEMPS.DAT', STATUS = 'OLD')
```

and the read statements would read the 100 records, *in the order written*, from the file, either in a loop that executes 100 times, or until the end-of-file is encountered (by using an END= branch in the READ statement).

## A Simple Direct-Access Program

A direct-access program would be slightly different. First of all, we would have to *specify* in the OPEN statement that this was to be a direct-access file, and we would have to specify the length of each record. Furthermore, we could *not* use list-directed I/O, as we did in the sequential access program.

```
OPEN (3, FILE = 'DTEMPS.DAT', ACCESS = 'DIRECT', RECL = 10,
& STATUS = 'NEW')
...
      WRITE (3, 33) TEMP
33  FORMAT(F10.3)
```

Since we did not specify record numbers in the WRITE, they were numbered in order from 1 to 100. If we had wanted to specify the record numbers, we would have had to include a REC= phrase in the WRITE statement, followed by an integer record number. These numbers do not have to be in sequential order.

We cannot use ENDFILE with a direct-access I/O operation, and, even though we were able to use a list-directed READ from the keyboard, we had to use formatted WRITES to the direct-access file. READs from this file in a second program would also have to use a FORMAT; we would have had to declare the file as direct-access and specify the record length when we OPENed the file.

We could have also used the option FORM= 'UNFORMATTED' to write this file, which option we will discuss next.



## UNFORMATTED INPUT/OUTPUT

So far, all of our files have been written out or read in as *formatted* files. This has the advantage that a formatted output file can be printed out for human consumption; but it is a time-consuming process, because all format statements that turn data into characters for output take considerable computer time. If the data you are creating in your program will only be used as input to another FORTRAN program, you could have it dumped out to the file in its raw *binary* form, which will require no time-consuming conversions; this is done using *unformatted* I/O. Unformatted I/O makes no use of formats, or of list-directed I/O (which also converts the data to characters). Thus, there are no format specifications or \*s in the format position in the I/O statements. Examples of unformatted I/O statements are:

```

OPEN (U1, FILE = 'ONE', STATUS = 'NEW', FORM = 'UNFORMATTED')
WRITE (U1) list
OPEN (U2, FILE = 'TWO', STATUS = 'OLD', FORM = 'UNFORMATTED')
READ (U2, END=25) list

```

An *unformatted* file may be either sequential- or direct- access. It is a bit more tricky to determine the record length (RECL=) for a direct-access file when using unformatted data. It will be processor-dependent, but probably based on the number of bytes in the record; if the data does not fill the record length given, the remainder of the record will be undefined. However, the record must be long enough to accommodate the data list written, so it is best to err on the long side in estimating the record length.



## UPDATING A FILE

To "update" a file is to make changes in its contents, which may include additions, deletions, and/or altered records. Since a sequential file is written out in order, you cannot make additions or deletions on the original file medium, and it is even dangerous to try to *alter* an existing record in place. Thus, the best method for updating an existing file (referred to as the "master file") is to read it from its external storage medium into memory (it may have to be read in pieces, if it is a long file), make the desired changes in memory, and then write the revised version of the file out to a new, updated file.

For example, let us say that in our sequential-access file of temperatures created earlier, we wanted to delete the 20th record, alter the value in the 25th record (update it), and insert a new record between the 49th and the 50th. We could do this by reading in all the data to an array, since it is not a huge amount of data, modifying the array, and writing out the array to a new file (TEMPUP). Rather than reading all 100 values from the file into the array, and then deleting entry 20, by moving all the array entries from 21 through 100 up one place, we will read in the first 19 records to the array, read but not store the 20th, and then read the rest. This is more efficient.

```

PROGRAM UPDATE
REAL TEMP, TEMPO(100)
OPEN (3, FILE = 'TEMPS.DAT', STATUS = 'OLD')
OPEN (4, FILE = 'TEMPUP.DAT', STATUS = 'NEW')
DO 40 I = 1, 19
 40      READ (3, *) TEMPO(I)
*           SKIP RECORD 20 TO DELETE IT
        READ (3, *) TEMP
        DO 50 J = 20, 48

```

```

50      READ (3, *) TEMPO(J)
*
*           INSERT RECORD AFTER #48 (OLD #49)
PRINT*, 'TYPE IN VALUE FOR NEW TEMP TO BE INSERTED 49-50'
READ*, TEMPO(49)
DO 60 K = 50, 100
60      READ (3, *) TEMPO(K)
*
*           CHANGE OLD RECORD # 25 (NEW RECORD # 24)
PRINT*, 'TYPE IN NEW VALUE TO UPDATE OLD 25TH RECORD'
READ*, TEMPO(24)
*
*           WRITE OUT UPDATED INFORMATION TO NEW FILE
DO 100 I = 1, 100
100     WRITE (4, *) TEMPO(I)
ENDFILE 4
CLOSE 3
CLOSE 4
STOP
END

```

A direct-access file can be modified in place. New records can be added as long as they have distinct new record numbers, and existing records can be modified in place (since they must all have the same record length, which was specified in the OPEN statement). A direct-access record cannot be deleted (though an entire file may be deleted), so the effective way to "delete" an unwanted file is simply not to access its record number when you are processing the data in the file. The alternative is to set some sort of a "flag" value in the file that indicates that it should not be processed (but then you must at least read it).

Thus, if we wanted to make the same modifications as we did in the sequential file to the direct-access version of the temperature file (remove record 20, update record 25, insert a record between old 49 and 50), we would proceed differently. First of all, we cannot insert a new record number between two numbers that are already sequential. Thus the program that wrote out the direct-access file should have been modified to number the records from 10 to 1000, not from 1 to 100, as follows:

```

NREC = 10
DO 30 I = 1, 100
    READ*, TEMP
    WRITE (3, 33, REC = NREC) TEMP
33      FORMAT(F10.3)
    NREC = NREC + 10
30      CONTINUE

```

Now to insert a record "between" the old 49th and 50th records (which are numbered 490 and 500, respectively), we need only write out a new record number, say, 495:

```
OPEN (3, FILE = 'DTEMPS.DAT', ACCESS = 'DIRECT', RECL = 10,
& STATUS = 'OLD')
PRINT*, 'ENTER VALUE TO BE INSERTED BETWEEN 49 & 50TH RECS'
READ *, TEMP
WRITE (3, 33, REC = 495) TEMP
```

To "update" the 25th record (number 250), we need only rewrite over it, in place:

```
PRINT*, 'ENTER NEW VALUE FOR 25TH RECORD'
READ*, TEMP
WRITE (3, 33, REC = 250) TEMP
```

However, "removing" the old 20th record cannot be done on a direct-access file (although the entire file can be deleted). Thus, this would have to be taken care of in the program which is to read and process the data from the file. It should simply read records numbered 10–190 (in steps of 10), records 210–490 (in steps of 10), record number 495, and records 500–1000 (in steps of 10). A record that is not processed is effectively deleted. The other alternative is to mark ("flag") the file to be "deleted" (that is, ignored), read it, and note the flag.



## INTERNAL FILES (CORE-TO-CORE DATA TRANSFER)

Data can be reformatted internally by treating character variables as if they were I/O devices. Thus, a character variable can be treated as the *input* unit in a READ statement, and its contents can be reformatted, using FORMAT conversion descriptors, to map into a list of variables. For example, if the character variable CC has been defined to contain a month, day, and year, stored such that the month name is in a 9-column field, followed by a blank, followed by a 2-digit integer date, a comma, a space, and a 4-digit integer year (all stored as part of the character string), it can be reformatted and split up into a character month, and integer date and year, in the following way:

```
CHARACTER CC*18, MONTH*9
INTEGER DAY, YEAR
CC = 'SEPTEMBER 17, 1990'
READ (CC, 5) MONTH, DAY, YEAR
5 FORMAT (A9, 1X, I2, 2X, I4)
```

Note that the character variable CC simply *takes the place of* an I/O unit in the READ statement. Thus, instead of accessing a record of character data from an external device and translating it according to the FORMAT, this statement takes an existing character variable and reformats it.

A list of data of mixed types can be *written* out to an external device (such as a screen or printer), by converting it to character form according to FORMAT descriptors. Similarly, a list of data can be reformatted into character form and “written” into a character variable instead of to the screen. For example, we could take a list of the integers from 1 to 31 and write them into a single character variable DATES, two digits (as characters) at a time:

```
CHARACTER DATES*62
WRITE ( DATES, 66) (ID, ID = 1, 31)
66 FORMAT (3I2)
```

Now, having these dates represented as characters is potentially useful. Recall the calendar problem (#19) at the end of Chapter 7, in which we wanted a character array of 31 entries, so that when we wanted the character equivalent of an integer that represented the date in a month, we could store it in the character calendar array. At that time, we suggested filling such a character array (DATE) by using a long DATA statement. Now we can do it much more simply, if we can figure out how to extract the 31 two-character strings from DATES, and store them into DATE in the proper order. This could be done using a loop which extracted successive substrings from DATES:

```
CHARACTER DATES*62, DATE(31)*2
J = 1
DO 20 I = 1, 31
    DATE(I) = DATES(J:J+1)
    J = J + 2
20 CONTINUE
```

However, it could be done even more simply, without requiring any loop or substring operations, just by equivalencing the array to the string; thus filling DATES is filling DATE at the same time:

```
CHARACTER DATES*62, DATE(31)*2
EQUIVALENCE ( DATES, DATE(1) )
```

Notice that the use of EQUIVALENCE is necessary to do this here, since the rules for internal data transfer do not allow us to reformat data and store it into a character *array*.

## ◆ WRITING TO PRINTER AND SCREEN IN THE SAME PROGRAM

On most systems, as you probably have discovered by now, your program PRINTs and WRITEs generally create output to the terminal screen when your program runs. If you want the program output to go instead to the printer, there is a mechanism by which you can redirect the output to a disk file, instead of the screen, and that disk file can then be sent to the printer. This arrangement generally works well, but it means that you can send your output to the screen, *or* to the printer, *but not both*. This restriction does not usually pose a difficulty, until you run up against a problem where you *need* screen information as the program runs (in, say, an interactive program to play tic-tac-toe or checkers), and you also want printed output. How can this be accomplished?

The way in which to output to the screen and the printer in the same run is to incorporate duplicate WRITE statements in your program. The first of these WRITE (or PRINT) statements will send output to the screen; the second will send the same information to an external file you have created (OPENed), which can be printed after the program is completed. Your system editor probably provides you with an instruction that lets you *copy* a line already in your program into another place in the program. Thus, you only have to copy the WRITEs (to \* or unit 6 or whatever the default unit for the remote output is on your system), or PRINTs, again on the line below, and then use the editing capability to alter the unit number (or a PRINT to WRITE ( ), if needed). This, and inserting OPEN and CLOSE statements for your external output file, will take care of the problem.

After you have inserted the OPEN statement:

```
OPEN ( 3, FILE = 'TICTAC.OUT', STATUS ='NEW')
```

you can go into the file and create the duplicate WRITE statements. For example, a copy of this line:

```
WRITE (*, 8) ((BOARD(I,J), J=1,3), I=1,3)
```

will create an identical line below it, in which all you have to do is go in and change the unit number to 3, so you have:

```
WRITE (*, 8) ((BOARD(I,J), J=1,3), I=1,3)
WRITE (3, 8) ((BOARD(I,J), J=1,3), I=1,3)
```

This method will work nicely for any program in which you want to create output both to the screen and the printer.

(Note: Some systems may have a sort of "Photo" capability, in which this command will allow you to send a copy of the screen to the printer; if your system has such a capability, you do not need to go through the duplicate WRITE statement modifications.)



## VARIABLE FORMATTING

Occasionally, situations may arise where you would like to be able to specify a *variable* value in a FORMAT statement (but this is not allowed in Standard FORTRAN). For example, you are writing a subroutine that will compute and print out, alongside the array, the average value of each row, for two-dimensional integer arrays of any size. You will use variable dimensioning, so that your subroutine can handle different-sized arrays. The arguments to the subroutine should be the array name (dummy name KAT), the number of rows (M), and the number of columns (N). When you print out the array, you will do that in usual uniform spacing, a row at a time. But when you print the row averages next to each row, you would like to space them a bit away from the array, so that they do not look like just another column in the array. Furthermore, since you want the array average to be a *real* value, it must have a different format descriptor than the rest of the array. The problem is, how do you set up the format for printing out the array and the row averages?

Since your subroutine is supposed to handle arrays of different dimensions, you cannot assume one repetition factor for the I format for the array values. Even if you assume you know the size (in columns) of the largest array you may get in the subroutine, you cannot just write the format for that. For example, suppose you know the largest array will have no more than 25 columns; you cannot just write the output format as:

```
33 FORMAT( / 5X, 25I4, 8X, F8.2)
```

since this will work fine for the 25-column array, but what will happen with smaller arrays? The list to output the row values, followed by the row average, will be of the form:

```
PRINT 33, (KAT(I,J), J = 1, N), ROWAVE
```

and if N is less than 25, it will then try to print the real row average with an integer format, and you will get an output error.

What we would *really* like to do is to be able to put a *variable* (N) right into the FORMAT statement, something like:

33 FORMAT ( / 5X, NI4, 8X, F8.2)	{illegal}
----------------------------------	-----------

but *no* compiler we know of will allow this. It is possible that your compiler may have a special, nonstandard way of handling this, however. We are familiar with two that allowed you to put a special character in the place where you wanted the variable value in the FORMAT statement, and then you could put the variable itself in the PRINT list, at the point where it would match up with the special character in the FORMAT. On one system we used (a CDC), this special character was an '='; on the other (a Burroughs/Unisys), it was a '\*'. Thus, on either of these systems, you would rewrite the output statement and FORMAT as:

```
PRINT 33, N, (KAT(I,J), J=1, N), ROWAVE
33 FORMAT(/ 5X, =I4, 8X, F8.2) {CDC}
or
33 FORMAT(/ 5X, *I4, 8X, F8.2) {Burroughs}
```

and the value of N would be implemented at execution time for the special character in the FORMAT. The special character could go any place in a FORMAT where you would normally use a constant; for example, =I5, I=, =X, T=, and so on. Check to see if your system has such a feature; it is very useful. It is sometimes referred to as a "run-time editing" feature.

Another system we used recently (a VAX/VMS system) had a different, nonstandard way of accomplishing the same job. It would allow you to put any integer variable or expression into a FORMAT statement, as long as it was enclosed in angle brackets—i.e., <>. Thus on the VAX, we would modify our FORMAT 33 to:

```
33 FORMAT (/ 5X, <N>I4, 8X, F8.2)
```

However, on many compilers such a nonstandard way of handling run-time editing may not exist. If you do not have such a special feature on your system, there is another way you can implement variable formatting without departing from Standard FORTRAN 77. We saw in Chapter 6 that you can write formatted I/O statements without actually using the FORMAT statement and its corresponding statement number. The method in which we defined a character variable, and then substituted that in place of the statement number of the FORMAT will be of use here. To handle the problem of variable formatting in our subroutine to average the rows of any integer two-dimensional array, we will begin by creating a character variable for the format that will initially leave the repetition factor on the I4 format for array elements blank (that is, unspecified):

```
CHARACTER C*35
C = '(/ 5X, I4, 8X, F8.2)'
```

Now what we want to do is to have our subroutine define the part of the character string where the repetition factor for I4 goes at execution time. We determine that this is the substring C(8:9). The only remaining problem is that the value of N (what we want as the repetition factor) is integer, and we must convert it to character. But this is not difficult, given what we have learned recently about internal data transfer. We know that the value of N will be a one- or a two-digit integer. All we have to do is to WRITE it into a character variable which we can then put into C(8:9).

```

CHARACTER C*35, REP*2
WRITE (REP, 44) N
44 FORMAT(I2)
C(8:9) = REP

```

Thus we see that we can create any variable format character string, and modify it appropriately at execution time. For the sake of completeness, we should write the subroutine we have been describing to average rows of any two-dimensional integer array.

```

*****
*          SUBROUTINE TO AVERAGE ROWS OF TWO-D ARRAYS      *
*****
SUBROUTINE AVEROW (KAT, M, N)
* THIS SUBROUTINE ACCEPTS ANY INTEGER TWO-D ARRAY (KAT) OF ANY
* SPECIFIED SIZE (M ROWS AND N COLUMNS), AND AVERAGES THE ROWS
INTEGER KAT(M, N)
REAL ROWTOT, ROWAVE
CHARACTER C*35, REP*2, D*35
C = '(/ 5X, I4, 8X, F8.2)
D = (''1'', 3X, I4, 6X, ''ROW AVERAGE'')
* FILL IN THE NUMBER OF COLUMNS AS REPETITION FACTOR IN FORMAT
WRITE (REP, 4) N
4 FORMAT(I2)
C(8:9) = REP
D(13:14) = REP
PRINT D, (KOL, KOL = 1, N)
DO 20 I = 1, M
  ROWTOT = 0.0
  DO 10 J = 1, N
    ROWTOT = ROWTOT + KAT(I,J)
  10 CONTINUE
  KAT(I, 1) = ROWTOT / N
  20 CONTINUE
END

```

```

10    CONTINUE
      ROWAVE = ROWTOT/N
      PRINT C, (KAT(I,J), J=1,N), ROWAVE
20    CONTINUE
      RETURN
      END

```



## SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

The *sign* of a numeric value can be forced to be output by inserting as SP edit specifier in the FORMAT; SS turns it off. The edit control nP *scales* values on input and output; on input with appropriate real or d.p. values, it *divides* the value input by  $10^n$  before storing; on output of reals using F format, it will *multiply* the stored value by  $10^n$  before output; on output of reals with E or D format, or G that converts to E, the mantissa is multiplied by  $10^n$  and the exponent output is reduced by n.

BN (blanks are "null") and BZ (blanks are zeros) editors can be put into input formats, and control the interpretation of blanks read by that statement until another B editor is encountered. They override the default system convention and even the BLANK= control set in an OPEN statement for a file.

You can *tab* to specified columns of an input or output record by using Tn; on output, do not forget that the record created by the format may begin with a carriage control character, so a Tn control will cause the following field to output beginning in column n-1 of printer or screen. You can use *relative tabbing* to tab right (TRn) or left (TLm) of the last position specified in the format.

External files may be *direct* or *sequential* access. This Access parameter can be specified in the OPEN statement for a file. The OPEN statement must specify the unit number the file is connected to, a name for the file, and usually its STATUS (the status default is 'UNKNOWN', which makes the file status depend on your system default). Optional specifiers that may be included are ACCESS= (direct or sequential [the default]), RECL= (record length, required if access is direct), FORM= ('FORMATTED' or 'UNFORMATTED'), BLANK= ('NULL' or 'ZERO'), ERR= (followed by a statement number to branch to if an error occurs on opening the file), IOSTAT=ivar (where the variable ivar can store a value to indicate *which* error occurred on opening the file).

An ENDFILE statement can be used to write a special end-of-file (EOF) record at the end of data output in sequential form. The REWIND command goes back to the beginning of the very first record on a sequential file, and BACKSPACE moves to the beginning of the immediately previous record (in a sequential file).

After you have finished reading from or writing to a file, you should CLOSE the unit to which it is connected (CLOSE u).

An INQUIRE statement can be used to determine the properties of a file; the "inquiry specifiers" may include ERR=, IOSTAT=, EXIST=, OPENED=, NUMBER=, NAMED=, NAME=, ACCESS=, SEQUENTIAL=, DIRECT=, FORM=, FORMATTED=, UNFORMATTED=, RECL=, NEXTREC=, and BLANK=, each of which should be followed by an appropriate type variable to contain the answer to the inquiry.

Generally, we write and read *formatted* files, but on occasion the information written and read is not for human consumption, but only to transfer data between computer programs; in such a case, it is more efficient to write and read *unformatted* (that is, binary), files, by specifying FORM = 'UNFORMATTED' in the OPEN.

Often we want to "update" a file, that is make changes, additions, and deletions. A direct access file may be modified in place, but a sequential file cannot be written over without danger, so that a new output file should be created for the modified data. On a sequential file, a deletion or addition should also be done by rewriting to a new file. On direct access, files cannot be deleted (though they can be skipped over or "flagged" to be ignored), but records can easily be added.

Core-to-core data transfer can be used to reformat data internally using READs to a character variable taking the place of a unit number, or WRITEs from a character variable to a list of variables, according to a specified format.

To have a program create output to both screen and printer, use duplicate output statements. Variable formatting can be accomplished by using special characters (\*, =) or markers (<>) on some systems (non-Standard), or by using a character variable (in which segments can be changed) to define the format description.



## EXERCISES

1. The series (of an infinite number of terms):

$$1 + 1/2 + 1/4 + 1/8 + 1/16 + \dots$$

converges to 2. Experiment with writing this series in single precision, first adding up 200 terms, then seeing how many terms it takes before the sum "rounds up" to 2.0 (set a limit of, say, 2000 terms, so you don't run too long). Now repeat the same experiment using double-precision variables and constants in your summation. Print out your results in double precision.

Now that you have determined how many terms it took to become equal to 2.0 or greater(!), try the same number of terms, but running the summation

in reverse order, from the smallest to the largest term, and see how much difference it makes.

2. Rewrite one of your large number problems from the preceding chapter using Iw.m format to get internal leading zeros in blocks to print out.
- 3. Write a program to calculate large Fibonacci numbers, using your large-number techniques. For example, print out  $F_1$  through  $F_{100}$ . Use Iw.d to print leading zeros in internal number blocks. Remember,

$$F_n = F_{n-1} + F_{n-2} \quad \{F_1 = F_2 = 1\}$$

4. Experiment with writing out tables of powers of 2 (both positive and negative powers) as real values, using F, E, and G formats. Which gives you the most satisfactory table? Can you extend your table by using double precision values? (The question here basically is, does double precision on your machine extend only the number of significant digits available, or does it also allow you to store reals of larger [and smaller] magnitudes?)

5. Using L formats, print out logical “truth tables” for the basic .AND., .OR., and .NOT. logical operations, and then for some more complex expressions such as (X.OR.Y.AND..NOT.(X.AND.Y))

Be sure to include your basic logical variables (each of which can take on two possible values) in your table. Use the tables to demonstrate the equivalence of DeMorgan’s Laws:

$$\begin{aligned} \text{.NOT. } (X \text{ .AND. } Y) &\equiv \text{.NOT. } X \text{ .OR. } \text{.NOT. } Y \\ \text{.NOT. } (X \text{ .OR. } Y) &\equiv \text{.NOT. } X \text{ .AND. } \text{.NOT. } Y \end{aligned}$$

The tables should look something like:

X	Y	X .AND. Y	.NOT.(X.AND.Y)	etc.
T	T	T	F	
T	F	F	T	
F	T	F	T	
F	F	F	T	

6. If you include more than two logical variables in your expression, remember that the number of lines in your table will be  $2^n$ , if there are n

variables in the expression. For example, set up a table to evaluate the expression:

$$A \text{ .AND. } (B \text{ .OR. } C)$$

for all values of A, B, and C. Compare it to the table of:

$$(A \text{ .AND. } B) \text{ .OR. } (A \text{ .AND. } C) \quad \{\text{Distributive Law}\}$$

**7.** Try tabbing (both relative and absolute) to create interesting output patterns, and be sure you understand how it works.

**8.** Using one of the techniques you have learned for finding prime numbers, write out the primes from 2 to 2000 to a sequential disk file. Use ENDFILE to mark the end of the data, rather than counting how many primes you have found. Then use this file in a program to determine the prime factors of any input value up to 2000. (See problem 15 in Chapter 7.)

**9.** Create a disk file of something dear to your heart—an alphabetical mailing list for Valentine's Day cards, a list of all of your records, or of all the movies you have on tape, a list of the birds you have sighted, a list of companies you want to apply for jobs with, or something else. Then write subroutines to aid you in easily *updating* the file—adding a record, deleting a record, or altering a record.

**10.** One method of creating record numbers for direct-access files is called "hashing," that is, taking some representative value for the record and making "hash" out of it, altering it in some way to create a (hopefully) unique record number. One method would be to take the first or last two or three digits of some representative integer value in the record. A more interesting method is to take the same integer value and create a hash code out of the leftmost two digits plus the rightmost two digits times 100; write a function to do this.

Another method is to take the integer, multiply it by some prime number (997, for example), add a prime (say, 113), and take the result mod 1000 (or mod 100 or mod 10000, or even mod some unusual number such as 771) as your record number. Remember that to take a number mod some base is to calculate the remainder when the number is divided by that base. The choice of base will bear a relation to the range of *possible* record numbers that can be generated. Write a function to do this.

Occasionally, hashing creates a "hash clash;" that is, there is already a record with that number. You can check for this by keeping an array of record numbers you have used so far. If a hash clash arises, use the next sequential record number (that is, the hashed value plus 1). Write code that will check for a hash clash and, if one occurs, come up with a new record number, and continue doing this until an unused number is found.

**11.** You are writing a program to access records that were written to a direct-access file using a hashing method. The records you want begin with a six-digit

variables in the expression. For example, set up a table to evaluate the expression:

$$A \text{ .AND. } (B \text{ .OR. } C)$$

for all values of A, B, and C. Compare it to the table of:

$$(A \text{ .AND. } B) \text{ .OR. } (A \text{ .AND. } C) \quad \{\text{Distributive Law}\}$$

**7.** Try tabbing (both relative and absolute) to create interesting output patterns, and be sure you understand how it works.

**8.** Using one of the techniques you have learned for finding prime numbers, write out the primes from 2 to 2000 to a sequential disk file. Use ENDFILE to mark the end of the data, rather than counting how many primes you have found. Then use this file in a program to determine the prime factors of any input value up to 2000. (See problem 15 in Chapter 7.)

**9.** Create a disk file of something dear to your heart—an alphabetical mailing list for Valentine's Day cards, a list of all of your records, or of all the movies you have on tape, a list of the birds you have sighted, a list of companies you want to apply for jobs with, or something else. Then write subroutines to aid you in easily *updating* the file—adding a record, deleting a record, or altering a record.

**10.** One method of creating record numbers for direct-access files is called "hashing," that is, taking some representative value for the record and making "hash" out of it, altering it in some way to create a (hopefully) unique record number. One method would be to take the first or last two or three digits of some representative integer value in the record. A more interesting method is to take the same integer value and create a hash code out of the leftmost two digits plus the rightmost two digits times 100; write a function to do this.

Another method is to take the integer, multiply it by some prime number (997, for example), add a prime (say, 113), and take the result mod 1000 (or mod 100 or mod 10000, or even mod some unusual number such as 771) as your record number. Remember that to take a number mod some base is to calculate the remainder when the number is divided by that base. The choice of base will bear a relation to the range of *possible* record numbers that can be generated. Write a function to do this.

Occasionally, hashing creates a "hash clash;" that is, there is already a record with that number. You can check for this by keeping an array of record numbers you have used so far. If a hash clash arises, use the next sequential record number (that is, the hashed value plus 1). Write code that will check for a hash clash and, if one occurs, come up with a new record number, and continue doing this until an unused number is found.

**11.** You are writing a program to access records that were written to a direct-access file using a hashing method. The records you want begin with a six-digit

I.D. number which is known to you. The I.D. number has been hashed into a four-digit record number by the following formula:

$$\text{record \#} = (\text{ID} \times 997 + 113) \bmod 10000$$

Write code that will access the correct record number for this I.D. number. However, hash clashes could have occurred in writing the file. You will be able to determine this because, when you access the record number, it contains the wrong I.D. The record contains I.D., name, rank, and age, formatted as:

```
FORMAT(I5, A20, 1X, A12, 1X, I3)
```

If this happens, keep trying record numbers one higher each time until you find the right I.D. number. Be sure to use the correct form of the OPEN statement to read this direct-access file.

- 12. Write a program which uses core-to-core internal data transfer methods to convert any integer value that is calculated or read in into a 12-character string. Then test the string to determine how many digits the integer contains. Compare this method with another routine you write to determine how many digits the integer has using numeric methods. Which is easier?
- 13. A method we will encounter in the Simulations chapter (von Neumann's mid-square method) involves squaring an integer (say it is a 4-digit integer) and then extracting the *middle* four digits (out of 8) of the result as your answer. Write a numeric subroutine or function that will do this for any 4-digit integer argument. Then write a function which will take the four-digit integer, square it, turn the result into a character string (using core-to-core transfer), extract the middle 4 characters of the string, and then rewrite this 4-character string as a 4-digit integer, also by using core-to-core. Which is easier?
- 14. Read in a text from disk, one 80-character line at a time, until an end-of-file is met (it is a sequential file). Now rewrite the text replacing all four-letter words with four stars (\*\*\*\*). If you *really* want to censor the file, only replace selected four-letter words you have stored in an array.
- 15. Read in a text from disk, as above. This time replace all occurrences of 'MAN' or 'WOMAN' by 'PERSON', and all occurrences of 'MEN' or 'WOMEN' by 'PERSONS'. Note that this will take a bit more thought, since the text should be adjusted to accommodate the replacement word, which is longer than the word it is replacing.
- ♦ • 16. Find a method of variable formatting that will work for you to print out a "plot" of a function of one variable (say, a sine curve, or cosine curve, or some such), by using the variable format to *tab* to the appropriate column of the output to print the function character. The plot will go down the page. Now use the same approach to plot two different functions at once.

17. You can get the effect of motion on your screen output by using the '+' carriage control. Begin with a simple print statement with a blank or '0' carriage control that puts you on a new line. Using variable formatting, create a loop in which you space over a varying number of columns and then print out some symbol or symbols (such as '///'). The loop counter can be used to determine the spacing before the symbol (for example, `<I>X` or `T<I>`, if you have that method of variable formatting; or `*X` or `T=` with the I value in the print list, if your system has that method; or construct a character variable for your format and alter the character(s) determining the number of spaces, if that is the only method available to you—as we did in the row-average program). On the screen, using a '+' carriage control to stay on the same line, and then spacing past your previous output will wipe out the previous output as it passes it, thus giving the effect of your symbols moving across the screen. If you run the same program to the printer, all of the characters output at different stages of the loop will appear on the output line (possibly with some overprinting).

- 18. The '+' carriage control will also allow you to create *overstrike* effects (to the printer) if you want them. To begin simply in learning to use this tool, write a program which will output your name to the printer and then underline the name.

19. *CORRELATION*. A science (physical or social science) often wants to determine whether there is some relationship between two variables (such as age and income, or height and weight, or temperature and elasticity, and so on). We thus would like a measure of how one changes as the other changes. A common correlation technique is the Pearson product-moment correlation ( $r$ ); it represents the extent to which the same individuals or events occupy the same position relative to two variables. If the set of measurements on each event is  $(x_i, y_i)$ , then the Pearson correlation  $r$  is defined as:

$$r = \frac{\sum (x_i - \bar{X})(y_i - \bar{Y})}{\sqrt{\sum (x_i - \bar{X})^2} \sqrt{\sum (y_i - \bar{Y})^2}}$$

where  $\bar{X}$  and  $\bar{Y}$  are the average values of  $x$  and  $y$ , respectively. Write a program to read in a set of measurement pairs  $(x, y)$  from a file until an END-OF-FILE is encountered, saving the  $x$  and  $y$  values in arrays. Calculate the average values for the two measurements, and then calculate and print out the Pearson correlation coefficient. Note that the Pearson coefficient is merely a measure of the *linear* relationship between the two variables; it will be close to +1.0 or -1.0 for a perfect linear relationship, and close to zero if there is no *linear* relation. However, a Pearson  $r$  value close to 0.0 does *not* imply that there is no relation between the variables, simply that it is not linear. The variables might have a perfect curvilinear relation and have an  $r$  of zero.

- ◆ 20. *Scattergram.* Since the Pearson  $r$  in the previous exercise only gives us a measure of linear relationship (or lack of it) for a pair of variables, it is often useful also to examine a *visual* representation of the relationship. One way to do this is by using a scattergram plot of the points defined by the measurement pairs. Take the same file described in problem 19, read in the  $x$  and  $y$  values to arrays, as in that problem, and then calculate the *ranges* for  $x$  and  $y$  (maximum – minimum). Use this information to fill a character two-dimensional array with the points defined by the measures. Since you cannot variably dimension your character array, dimension it initially to be fairly large (say, 101x101); then *scale* your  $x,y$  values to be points in the array. For example, if  $xrange$  ( $xmax - xmin$ ) comes out to be 50., each  $x$  value relative to  $xmin$  should be multiplied by 2, rounded (up or down), as appropriate, and stored as a point in that column of the array. (Of course, the character array should first be initialized to all blanks.) If  $xrange$  and  $yrange$  are the values you have calculated as the ranges of the two variables, then perform the scaling from a  $y$ -value to a row coordinate as:

$$\text{row} = \text{IROUND}( 100/yrange * (y_i - ymin) )$$

and similarly, the column coordinate should be calculated as:

$$\text{column} = \text{IROUND}( 100/xrange * (x_i - xmin) )$$

You can use the IROUND statement function you wrote for problem 6 in Chapter 9. The character array should be dimensioned:

```
CHARACTER SCATTER(0:100, 0:100)
```

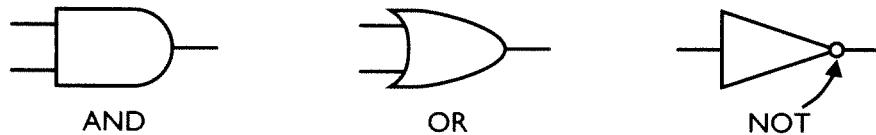
and filled with blanks to begin. Then when a (row,column) pair has been determined, a point (say, '\*') should be stored in that position of the SCATTER array. When you have done this for all of the measurement pairs from your file, you can then print out the SCATTER array to display your scattergram. It should be printed a row at a time, with a 101A1 format. Since the smaller values of  $y$  are near the beginning of the array, and they should really be plotted near the bottom of the picture, you should probably print the array beginning at row 100 and working down:

```

DO 30 I = 100, 0, -1
      PRINT 33, (SCATTER(I,J), J = 0, 100)
33  FORMAT(' |', 101A1)
30  CONTINUE
      PRINT 34
34  FORMAT(' |', 101('_'))
```

*Note:* The following two problems are optional. They relate to an area of special interest to many engineers and physicists, that of digital circuit design. The descriptions of the problems are long, since they include discussion of what Karnaugh maps are, and the problems themselves are complex. They are included for those who have an interest in this area.

**21.** Logical, or Boolean, analysis is the basis for digital logic circuit design. Such circuits are built up out of AND, OR, and NOT gates:



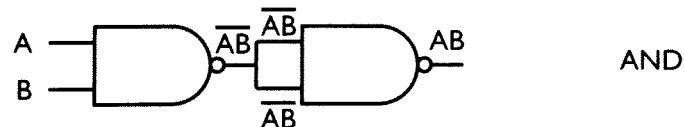
or NAND (not AND) and NOR (not OR) gates:



Thus a logical (Boolean) expression can be written for any circuit composed of such elements, and a circuit can be built to embody any logical (Boolean) expression. The notation is more often in terms of 1s and 0s, rather than Ts and Fs. Let us simplify our notation somewhat, so that we will write A.AND.B as AB, A.OR.B as A+B, and .NOT.A as  $\bar{A}$ . It can be shown that NAND and NOR gates are “universal”; that is, any of the basic logical connectives AND, OR, or NOT, can be implemented only using NAND gates (or only using NOR gates). And we know that any logical (Boolean) expression can be built up out of AND, OR, and NOT, so any logical expression can be built up out of just NAND gates (or just NOR gates); they are basic circuit building blocks.

Using your knowledge of logical expressions, and the form of the NAND and the NOR gates (expressions), prove to yourself that AND, OR, and NOT can all be implemented using only NAND gates. Then repeat the exercise using only NOR gates.

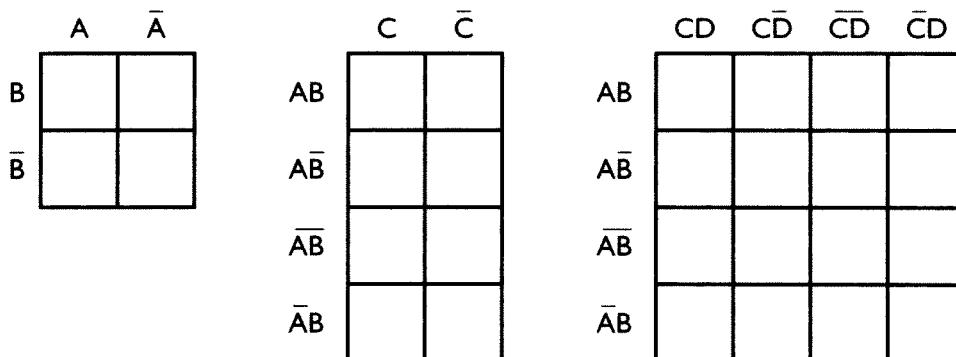
*Hint:* To begin with, notice that one NAND gate output, if the inputs are A and B, is  $\bar{AB}$ ; what if this output is then “split” and fed in as both inputs to another NAND gate (that is, you are taking not ( $\bar{AB}$  and  $\bar{AB}$ ) ). But  $\bar{\bar{AB}}$  and  $\bar{\bar{AB}}$  is  $\bar{AB}$ ; and not  $\bar{AB}$  is simply AB. Use some similar approach (remember you can split an input to be both inputs) to show that OR and NOT can also be built up with NAND gates. Now do the same for NOR gates.



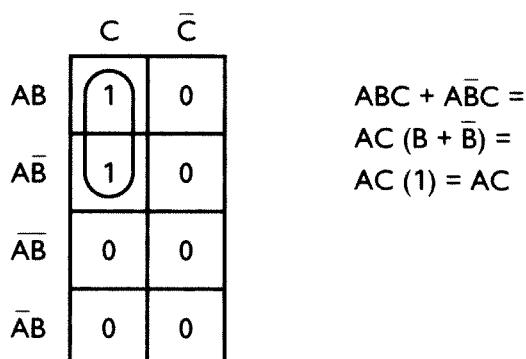
You may find it useful to define statement functions to do NAND and NOR operations. Then you can feed different inputs into these statement functions and test the results.

(Reference any digital logic text for help; for example, try Ronald J. Tocci, *Digital Systems: Principles and Applications*.)

**22.** It is often desirable to *simplify* a Boolean expression to a form that accomplishes the same result using fewer operations. This can save time and money in logic design. One way to do this, for an expression containing four variables or less, is that of the Karnaugh map. If an expression is written as a sum of products (that is, *or-ing* together a number of *ands*), and it can be proved that all expressions can be formulated in this manner, it can be *diagrammed* in a Karnaugh map, and this diagram can be used to find a simpler form for the expression. Karnaugh maps for expressions in two, three, and four variables are shown following. Recall that a variable A may appear as  $\bar{A}$ .



These diagrams can initially be considered to be filled with all 0s (Fs), and then a 1 placed in each position where a term appears in the expression that is to be simplified. We see that if a pair of 1s appear adjacent to one another, a simplification can be made, because for one of the terms both the term and its negation appear in the expression:



We can thus see that the term in the expression which appears as both  $B$  and  $\bar{B}$  can be dropped, cancelled out, to simplify the expression. Whenever two 1s are adjacent in the diagram, a term can be removed. The pair can occur either horizontally or vertically, and it can occur across boundaries (that is, the bottom term ( $\bar{A}B$ ) is “adjacent” to the top term ( $AB$ ), since they have one common term and one term that appears in both positive and negative form (all adjacent terms in the diagram satisfy this condition). Thus, in the two following, we can simplify:

	C	$\bar{C}$
AB	1	0
$\bar{A}B$	0	0
$\bar{A}\bar{B}$	0	0
$\bar{B}$	1	0

	C	$\bar{C}$
ABC + $\bar{A}BC$	0	0
= BC	1	1
$\bar{A}B$	0	0
$\bar{B}$	0	0

Isolated 1s represent terms which cannot be simplified.

There may be adjacent groups of four (which can be an entire column in a 3-variable map or a grouping in a 4-variable map), and these remove *two* terms that appear in both forms. These four-groups may also occur across boundaries.

	C	$\bar{C}$
AB	0	1
$\bar{A}B$	0	1
$\bar{A}\bar{B}$	1	1
$\bar{B}$	0	1

	CD	$C\bar{D}$	$\bar{C}D$	$\bar{C}\bar{D}$
AB	0	0	0	0
$\bar{A}B$	0	1	1	0
$\bar{A}\bar{B}$	0	1	1	0
$\bar{B}$	0	0	0	0

$$\begin{aligned} & \bar{A}BC + ABC + A\bar{B}C + \bar{A}B\bar{C} + \bar{A}BC \\ &= \bar{A}BC + C \end{aligned}$$

$$\begin{aligned} & \bar{A}BCD + ABCD + A\bar{B}CD + \bar{A}B\bar{C}D \\ &= BD \end{aligned}$$

There also may occur adjacent sets of eight 1s in a four-variable map. As you might be able to see by induction by now, this means that three of the four variables involved can be dropped out. These groups of 8 (octets) can also

occur across boundaries of the Karnaugh map. You (or the computer) must find the single variable that is in the same form in all 8 positions.

	CD	$CD$	$\bar{CD}$	$\bar{C}\bar{D}$
AB	1	1	1	1
$A\bar{B}$	1	1	1	1
$\bar{A}B$	0	0	0	0
$\bar{A}\bar{B}$	0	0	0	0

$$\begin{aligned} & ABCD + ABC\bar{D} + A\bar{B}CD + AB\bar{C}D \\ & + A\bar{B}C\bar{D} + A\bar{B}CD + A\bar{B}CD + ABCD \end{aligned}$$

$= A$

	CD	$CD$	$\bar{CD}$	$\bar{C}\bar{D}$
AB	1	0	0	1
$A\bar{B}$	1	0	0	1
$\bar{A}B$	1	0	0	1
$\bar{A}\bar{B}$	1	0	0	1

$$\begin{aligned} & ABCD + A\bar{B}CD + A\bar{B}CD + \bar{A}BCD \\ & + A\bar{B}C\bar{D} + A\bar{B}CD + A\bar{B}CD + \bar{A}\bar{B}CD \end{aligned}$$

$= D$

This Karnaugh map technique presents an interesting challenge for your programming talents. First you should figure out a way to accept "sum-of-product" expressions to be simplified (you may want to adopt a different notation for the "not"s, or complements, such as  $ABCD$ , since your keyboard input must all be on one line—perhaps a dash (-) preceding the variable to be negated, so that  $ABCD$  would be entered as  $-AB-C-D$ ). Then you should construct a two-dimensional array of the appropriate size for the number of variables involved, and figure out a relation between the variable names, and their negations, and the row or column they are assigned to (this assignment may help you later in deciding how to simplify the expression).

After the array is established (a subroutine would be able to variably dimension an array), initialize it to zeros (or Falses), and then fill in the 1s (or Trues) in those positions which represent a term in the expression. Then you must figure out a way to determine if there are groups of two, or of four, or even groups of eight in a 4-variable map. If such groups are found, determine how they are to be simplified (and once they are simplified they can be ignored). Be sure to include the terms represented by any isolated 1s in your final expression. Print out the final simplified expression.

It would be valuable in developing and debugging the program to figure out a way to display the Karnaugh maps; this will help you check your computer simplification, since by now you know how to do it by hand.

Do not worry about overlapping groups (pairs, fours, or octets)—each such group generates a term in the final expression. However, perhaps you should be careful, if there are any overlaps, not to destroy one after you have taken

care of another in the overlap (that is, for example, if you just set the elements of a quartet you had simplified to 0, it would wipe out the previous overlap it had with a different pair).

Notice that this is not a short or a simple program (fair warning), but it might make a challenging term project.

# CHAPTER 12



# VISUAL OUTPUT

Much scientific data is too complex to be comprehended in its numeric form. The answer to this problem is “scientific visualization”—making the information visual, in graphical form. This chapter introduces the programmer to methods of creating graphic output from a FORTRAN program to the printer or the screen. It includes interesting applications such as Cartesian graphs, biorhythm plots, and fractals.

During a solar eclipse, the sun's corona shines brightly around the moon's shadow.

*"Vision is the art of seeing things invisible."*

- Jonathan Swift, Thoughts on Various Subjects (1711)



## THE PRINTER AS PLOTTER

There are many graphics packages currently available that will run on input from your program or a data file you have your program construct, and they can drive different graphics output devices. However, you may not have immediate access to such a package, or you may merely want some fast, but pictorial, output directly from your program. In such a case, you can use either the terminal screen or the printer as your output device. To accomplish such output, you will have to think in terms of outputting characters to one of these two devices.

### Histograms

You may on occasion generate a table of *counts* of how many values in the data you are processing fall into different specified ranges. You could then just print out this table of counts, but a *histogram* of the distribution would be much easier to read and interpret. We have written a short program to read in 500 grades that lie in the range from 1 – 100, and we want counts of how many of these grades lie from 1 to 5, how many from 6 to 10, ..., and how many from 96 to 100. Rather than put in 20 IF tests to determine which counter to increment for a particular grade, we will perform a sort of "hash coding" that will *map* a particular grade onto the appropriate counter in the array of counters. We want a grade between 1 and 5 to increment counter number 1, a grade between 6 and 10 to increment counter 2, and so on. If we subtract 1 from the grade and divide the result by 5, using integer division which will truncate, and then add 1 to the result of the division, we will get the mapping we want.

Once the array of counts has been filled, we want to print out the appropriate number of asterisks (\*) on each line representing the range of grades. We will assume that no count will exceed the space provided by our output device for this problem. If it might exceed the available space, we would have to test for this, and scale down the number of \*'s printed by a factor of 2 or 3 to get them to fit. If variable formatting were available (see Chapter 11), we could use that to print out the number of stars to match the count, but it is not standard and may not be available on your machine. Another technique is simply to have a character array filled with stars, and use the count to determine how many elements of the array to print. This is what we have done in the program.

```
***** HISTOGRAM OF GRADE DISTRIBUTION *****
INTEGER COUNT(20), GRADE
CHARACTER HISTO(100)
DATA HISTO/L00**'/
***** INITIALIZE COUNTERS *****
DO 10 I = 1, 100
10   COUNT(I) = 0
***** READ IN GRADES AND COUNT *****
DO 20 K = 1, 500
   READ*, GRADE
   N = (GRADE-1)/5 + 1
   COUNT(N) = COUNT(N) + 1
20 CONTINUE
***** DISPLAY GRADE RANGES AND HISTOGRAM *****
PRINT 25
25 FORMAT('1',40X,'HISTOGRAM OF GRADE DISTRIBUTION'//' GRADE')
N = 1
DO 40 I = 1, 20
   PRINT 35, N, N+4, ( HISTO(J), J = 1, COUNT(I) )
35   FORMAT ('0', I2, '-', I3, 2X, 100A1)
40 CONTINUE
STOP
END
```

If the counts were too high to fit to the output device you are using (for example, the screen), you might want to print a star for every 2 values in your count, for instance:

```
PRINT 35, N, N+4, (HISTO(J), J = 1, COUNT(I)/2)
```

You might also replace the array of stars entirely, and make your output statement of the form:

```
PRINT 35, N, N+4, ('*', J = 1, COUNT(I))
```

This is simpler, since it saves any array references to HISTO.

## Plotting with Scaling

We will deal here with the mechanics of plotting a graph or similar picture to the printer, a picture which could also go to the screen if desired. Of course, our normal screen is usually about 80 columns wide, whereas a printer connected to a mainframe will usually accommodate 132 columns. Thus, the

plots we put out to a screen will be more limited in dimension—80 columns by roughly 25 lines, if we want to see the entire picture at one time. There are ways to do true graphics on a high-resolution screen, say of a microcomputer, but this involves delving into assembly language and extensions to the operating system, and is beyond the scope of our book. Computer graphics can fill a whole course in itself; we will restrict ourselves to simple plotting here, which can be very useful for the scientist.

We should first consider the “canvas” on which we can paint our picture. It will be roughly 25 rows by 80 columns, if we want to fill one terminal screen, or roughly 66 or 88 rows by 132 columns to fill one printer page. A printer page is 11 inches long, and the printer may be set up to print either 6 or 8 rows per inch. Eight rows per inch is more economical, but also more compressed. There are usually 10 columns per inch across a page from the printer. A typical screen will display slightly less than 4 rows per inch, and roughly 8 columns per inch. We will need to keep these proportions in mind when designing our pictorial output for either device.

We can compose our “picture” out of any of the available symbols on the system we are working with. Probably the character we will use most frequently is the blank, especially if we are plotting Cartesian graphs. If you think about it, when you plot an equation or a set of data points on prelined graph paper, the paper you begin with is totally blank (if you ignore the reference blocks), and you usually only fill up a small portion of its surface with your plotted points. We will begin our discussion with printer plots where the scale does not have to be the same on both axes, and then we will undertake plots of different Cartesian curves on paper or screen.

## Plots with Incommensurate Axes

Not all plots we will create need the same scale on both axes. If they do not, our task is much simpler. For example, if we are plotting distance or some other magnitude against time, there is no reason that a unit on one axis need have any relation to a unit on the other axis. Thus we should simply design such a plot so that it fills the output device nicely, and so that the spacing between rows plotted gives a pleasing picture—one that is not too cramped.

We might also have to work with input data, which might be given in some random fashion, instead of in the order we want to print out the plot. In such a case, we would probably fill a two-dimensional array with the data points as they are read in, and then print it out a row at a time.

The first crucial thing we must determine is the *range* of the values our plot will take on. Thus, either analytically (if we know ahead of time the nature of the data we are going to plot), or by searching the data, we must determine the minimum and maximum values across the page (the x-axis) and the minimum and maximum values down the page (the y-axis). From these, we can

determine the X RANGE (XMAX - XMIN) and the Y RANGE. The range of x-values must be spread enough across the screen or printer page to get good resolution. Generally, the more you spread it out, the better the picture will look.

Imagine, for example, that you want a plot of temperature as a function of time, from time = 1 to time = 121 (seconds), according to the following function:

$$\text{TEMP}(T) = 50 + 10 * \text{SQRT}(T)$$

From this, we know that temperature will range from 60 to 160 degrees (Fahrenheit). Our time axis will go in uniform stages, so we will make that our y-axis (down the page or screen) and the temperature our x-axis. We will want a time scale down the left margin, and we may want a temperature scale across the top (we will worry about that shortly). If we are going to the printer, we will probably have 132 columns to work with, which means we could easily allow 1 column for each degree of temperature (the simplest choice, which we will make). The values will range from 60 to about 160 degrees, so we will spread our temperature values over 101 columns. We could plot a point for every second time difference, which would give us 120 lines of output. We will try that first, and if we don't like how it looks, we can change it to plot every 2, 3, or 4 seconds.

Since scaling is not really a problem, we can commence with our program. But one concern remains—how do we do a plot? We suggested one method, using tabbing and variable formatting, in an exercise at the end of the previous chapter; but there are easier ways to do this. Let us view our plot as mostly empty space into which we are going to put a few points to plot. This suggests naturally that the best medium to work in is character variables. For our problem, let us define a character array LINE with 101 locations, for the 101 columns our graph will cover. It should be initially filled with blanks, and then we can put the plot point in each time when it is calculated. We have to relate a calculated temperature to an array position in LINE, but that is simple; it should agree with the following mapping:

TEMP	60	160
	↓	↓
LINE	1	101

This is an easy transformation to write. The LINE subscript should be the temperature -59 (to the closest integer). Thus, we will compute a temperature from the formula (a real), and then round it up or down to the closest integer (we wrote a statement function to that once; it is time to dust it off and make use of it again).

We will take, say, 5 spaces on the left of the page, then print the time with an I3 format, and space two more blanks to the plot itself. Thus the plot (array

## 438 VISUAL OUTPUT

LINE) will go from column 11 through column 111 on the printer. If we want to label the temperature axis, we could do it from 60 to 160 in steps of 10 across the top of the plot. If we print each integer label with an I3 format, we should then have 7 blank columns between each printing, and we should begin in column 9 (the 3-digit labels will be centered over the column they characterize). We are ready!

```
*****
*          TEMPERATURE PLOT PROGRAM
*****
*****  
      INTEGER T, TAXIS, TEMP, IROUND, LPOS  
      CHARACTER LINE(101)  
*** STATEMENT FUNCTION TO ROUND UP OR DOWN TO NEAREST INTEGER ***  
      IROUND(X) = X + 0.5  
*** STATEMENT FUNCTION TO CALCULATE TEMPERATURE AS FN OF TIME ***  
      TEMP(T) = IROUND (50. + 10.0*SQRT(FLOAT(T)) )  
      PRINT 3, (TAXIS, TAXIS = 60, 160, 10)  
      3 FORMAT('1',45X, 'TEMPERATURE VS. TIME'//5X,'TEMP',  
      & 1X, 11(I3,7X)// 5X,'TIME'// )  
      DO 20 T = 1, 120  
      DO 10 L = 1, 101  
      LINE(L) = ''  
10      CONTINUE  
      LPOS = TEMP(T) - 59  
      LINE(LPOS) = '*'  
      PRINT 11, T, LINE  
11      FORMAT(6X,I3,2X,101A1)  
20      CONTINUE  
      STOP  
      END
```

Now this program could be modified to print fewer time points just by changing the first DO statement to read:

```
DO 20 T = 1, 120, 2
```

or with a step size of 3, or 4, and so on. The program could also be modified to print out a "filled" curve, that is, asterisks (stars) from the left margin to the data point, by filling the array from position 1 to LPOS with asterisks:

```
DO 15 L = 1, LPOS  
15      LINE(L) = '*'  
      
```

To change this program to go to the screen instead of the printer, we have to scale the temperature axis differently. Since there are only 80 columns across on a normal terminal screen, we must scale down our temperature values to less than one degree per column. Since we want a time label on the left of the plot, which will take 3 columns, let us suppose we use the remaining 77 columns for the plot. We are then going to spread temperature values from 60 to 160 (101 values) over 77 columns. This means, in addition to subtracting 59 from the calculated temperature, we have to multiply the values by a scale factor. We want the following correspondence:

TEMP	60	160
	↓	↓
LINE	1	77

What is needed is a simple  $y = ax + b$  transformation. This can be obtained by dividing the *range* of values for the line subscripts by the *range* of values for the variable to be plotted (temperature), to get 76/100, as a multiplier, which we will then use to scale the variable value translated to zero (that is, by subtracting the low value, 60), and adding 1, since the shifted "zero" value of the variable will go in LINE position 1:

$$\text{subscript \#} = (\text{subscript range/var. range} - \text{low value}) + 1$$

Note that we add 1 because the first line subscript position (the one corresponding to the variable translated "zero" value) is 1.

We still want the temperature scaled value to be rounded up or down to the nearest column, so we modify as follows:

```

REAL SCALE, TEMP
TEMP(T) = 50.0 + 10.0*SQRT ( FLOAT(T) )
SCALE = 76.0/101.0
...
LPOS = IROUND ( SCALE * (TEMP(T)-60.0) ) + 1

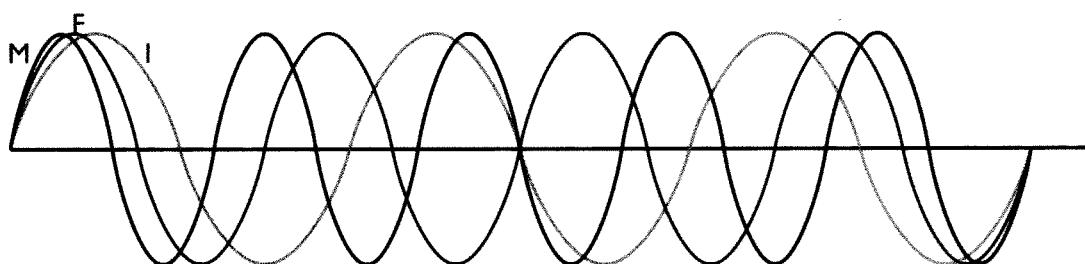
```

and the program will plot the curve to the screen.

## Biorhythms

An interesting exercise for you to do is to plot biorhythms. It has been speculated that the characteristics of a person's life fall into three categories that cycle through periodic ups and downs, like sine curves, throughout your life. These cycles are of different lengths, and they all began the day you were born. The "male" cycle has the characteristics of strength, courage, and physical prowess, and completes its up-and-down sinusoidal cycle in 23 days.

The “female” cycle has the characteristics of friendliness, creativity, and sensitivity, and it repeats every 28 days. The “intellectual” cycle is 33 days long, and has the obvious characteristics implied by the name.

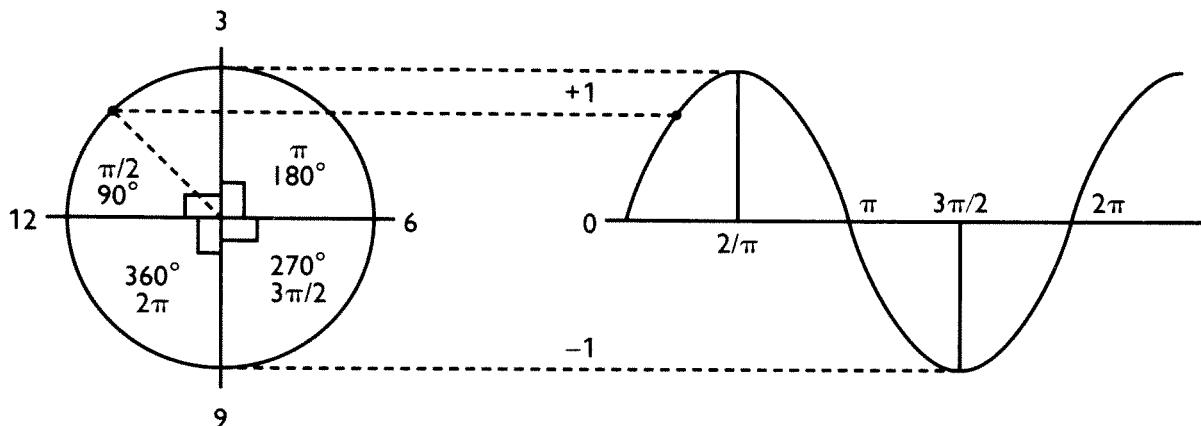


Days on which a curve crosses the axis (that is, changes from high to low, or low to high), are called “critical days,” and you are supposed to be particularly accident-prone on those days. If two of the curves cross at the same time, watch out! And if three curves cross at roughly the same time, friends suggest that you begin writing your will!

Some people take these biorhythms seriously; there are trucking companies and airlines that will not send their drivers or pilots out on days the biorhythms indicate are bad. In any case, it is an interesting exercise to amuse you and your friends (and though pseudo-scientific, you can learn much from it about plotting various scientific curves, and enjoy the exercise).

To plot a person’s biorhythms, you need to know the date on which they were born, and the current date, from which the biorhythms are to be plotted. We will set up the program so a varying number of days for the plot can be chosen. You already have written a program to determine how many days old a person is given their birth date and a current date; this program can just expand on that. For each day of the plot, you will need to find out at what point of the sine curve (cycle) the person is for each of the three cycles. This can be done by first determining how many days into the cycle the person is (done by removing all completed full cycles and seeing how many days remain—use the good old MOD function).

You can think of a sine curve as representing the motion of the hour hand as it sweeps around a clock, where a “full cycle” is “full circle.” Thus, at three o’clock, the sine curve has reached its peak, at 6 o’clock it is crossing the axis, at 9 o’clock it has reached its low point, and at 12 o’clock the cycle is complete. The magnitude of the sine curve at any time during the cycle is the sine of the angle which represents the portion of the full cycle (circle) swept out so far. A full cycle is 360 degrees, or  $2\pi$  radians. On the clock, if the hour is H, the angle swept out is  $H/12$  times  $360^\circ$  or  $H/12$  times  $2\pi$  (for an angle expressed in radians). At 3 o’clock,  $3/12$ , or  $1/4$  of the cycle has been completed, and the sine of  $90^\circ$  is 1 (the maximum value of the sine curve, which ranges between +1 and -1).



Also look at the projection of the hour hand onto the curve.

Now for our purposes, we will look at a full cycle as being represented by  $2\pi$  radians, since the SIN function in FORTRAN expects a *real* argument expressed in radians. For each of the three cycles of the biorhythms, you will calculate the fraction of the cycle completed on the given day (the number of days into the cycle, calculated by using MOD, divided by the length of the cycle); then if you take the SIN of this expressed as an *angle* (that is, the fraction times  $2\pi$ ), you will have the magnitude of the sine curve for the cycle on that day.

Once you have the sine curve magnitudes (between  $-1$  and  $+1$ ) for all three cycles for that day, your next problem is how to plot them. First of all, since you want the program to run for different periods of time, you should make the time axis run *down* the page, so you can vary its length. You should also print the dates (in the form mm/dd/yy) down the left margin as labels. This means that the only *scaling* you have to do is to spread out the sine curves over most of the printer page (or screen), and decide whether to plot dates every line or every other line (to make a nicer-looking plot, skip a line between the days on the time axis).

6/28/91	F	M	I
6/29/91		FM	I
6/30/91	M	F	I
7/ 1/91	M	F	I

The values you get from the SIN function for the three curves lie between  $-1$  and  $+1$ . You want to scale them so that they are spread over, say, 101 or 121 columns for the printer, or perhaps 71 columns if you are going to the screen. Thus, you want to figure out a transformation for the following relations:

$$\begin{array}{cccc}
 \text{SINE} & -1. & 0. & +1. \\
 & \downarrow & \downarrow & \downarrow \\
 \text{LINE} & 1 & 51 & 101
 \end{array}$$

Given the scaling we have done before, this should not be very difficult for you. You should use three different symbols for the three different curves (say, 'M', 'F', and 'I') to distinguish them from one another. Of course, you can only plot discrete points on a computer output; but after the graph is printed out, you can "connect-the-points" with pen or pencil.

A debugging hint for this program. If all of your points end up on the axis, you are probably getting integer truncation somewhere. Be sure that when you take the SIN of an angle, it is a real argument, and hasn't been truncated to zero because you divided an integer number of days into the cycle by an integer cycle length.

## Cartesian Graphs

In this section, we will discuss a similar approach to plotting using the computer, that of creating a line of character output and then sending it to the output device, creating a new line and sending that, and so on. If we are dealing with a printer, or with a screen without using special graphics tools that allow accessibility of any point on the screen, we are restricted to being able to output one line at a time, from top to bottom. We must keep this restriction in mind when we are designing a program to output graphical data. We will design our Cartesian plots one line at a time, beginning at the *top* of the graph, since the first line that we output is the one that will be at the top of the finished product.

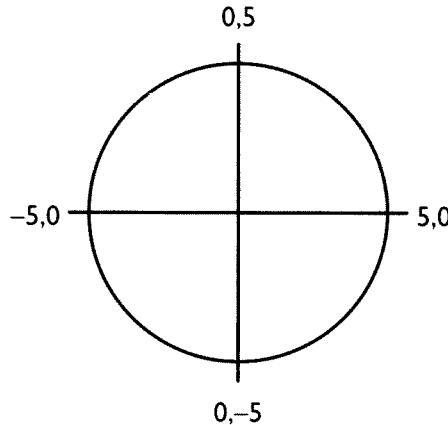
Before we jump right into our plotting process, we should back off to get some perspective on the nature of the graph we want to plot, the range of values we wish to include (since any Cartesian graph is really part of an infinite plane), and how big we want the resulting plot to be. This last feature will be conditioned by whether we are going to a printer or to the screen, since the device will determine available size and resolution, but we must also decide how much of the printer page or screen we want our plot to take up. Generally in these cases, the bigger the better, since the larger we make the plot, the better resolution we can get. After all, we will probably be plotting points on what is a continuous function, and we will be limited to being able to place them only at row and column intersections on our output device.

We will consider the plot of a simple Cartesian graph, which will exhibit enough features we will need to concern ourselves with to generalize well to any Cartesian graph. We want to plot a circle described by the equation:

$$x^2 + y^2 = 25$$

This will involve all four quadrants of the Cartesian plane, and it will be symmetrical about the origin. The range of values we want to print out is determined by the curve itself. By examining the equation, or a rough sketch of its graph, we discover that it intersects the x-axis at +5.0 and -5.0, and it

intersects the y-axis at +5.0 and -5.0. Other considerations are that we will want to display both axes, as in the sketch, and we would like nicely laid out scales to label both axes.



We will set up the program with certain presuppositions. We will begin by visualizing the plot a line at a time, beginning at the top of the picture. Initially, our clear sheet of "graph paper" (screen or printer) will be composed of lines filled with blanks. Then we will worry about putting in the x- and y- axes, so that the plot will have proper referents. Finally, we will actually plot the appropriate point or points for the curve (or curves) involved for the line we are constructing. Once the line has been properly filled, we will send it to the output device, and then move on to the next line.

We thus begin with each line as a "tabula rasa," a blank slate—a character array filled with blanks. We could have used a single character variable with the appropriate number of characters to fill the line, but we find accessing elements of a one-dimensional array slightly simpler than accessing one-character substrings of a character string. Next, we address the matter of axes. The y-axis will be in the middle of the plot, and the x-axis will occur halfway down the plot, horizontally across the page.

We have better resolution in the x-direction (across the page or screen) than in the y-direction, since there are more columns per inch than rows per inch on both the printer and the screen. Thus, we begin with uniformly spaced y-values, and calculate the corresponding x-value from the equation(s). The calculated x-value must go into a particular column, to be determined by truncation or rounding, but at least we have finer discrimination possible in this direction.

The next problem is that of scale. We might not have the same difficulties on many plots we might attempt, if the two axes represent totally different quantities, such as time versus some productivity measure, or height versus weight. However, in a Cartesian plane, we are looking at axes that should have the same scales. That is, if we consider this plane as representing spatial distances, a unit distance in the x-direction should be exactly as long as a unit

distance in the y-direction. If we look at the spacing available on our printer, the solution is not obvious, since we have 8 (or 6) rows per inch and 10 columns per inch. Scaling to the screen is simpler, if the screen has 4 rows per inch and 8 columns per inch, since these are nicely related in the ratio 1:2—we can scale to use two columns for each row. That is, whatever value range we assign to two adjacent rows, that same value range will occupy two columns in the x-direction.

The printer is not quite so neatly related regarding scale. We approach this problem by looking at the overall ranges of values we wish to plot in both the x- and y- directions, and then adjusting the number of rows and columns we will use to have a compatible scale. For example, our circle will have y-values that range from +5.0 to -5.0, and x-values that also range from -5.0 to + 5.0. Thus, if we calculate the *y-range* and the *x-range*, both will be equal to 10.0 in this case. If we spread our x-range over 125 columns (an odd number of columns, so that the middle column can be the y-axis), it will take up 12.5 inches (since there are 10 columns per inch across the printer). Thus, each x-unit distance on the plot will occupy 1.25 inches.

To make our y-units also 1.25 inches tall, we should spread our 10 y-unit range over 12.5 inches, and since we have 8 rows per inch, this would give us 100 interior distances in the y-direction. To make the plot more readable, since we intend to put a y-value on each line plotted, we will plot lines of the graph every other row. Thus we will have 51 different y-values (the middle one for  $y = 0$ , the x-axis) for our plot. It is simple to divide up the y-range into 50 segments, and we do that, storing an array of y-values, and then computing the corresponding x-values from the formula. We can simply determine the y-value for each new line, compute the x-value, and plot the point(s).

Given that we intend to spread the 10 x-units over 100 columns, the next problem is how to convert an x-value in the range from -5.0 to + 5.0 into a column number from 1 to 125. If we line up the correspondences we want, it is simple to determine the conversion equation. We clearly want the x-value of -5.0 to go in the leftmost column, or column 1, the x-value of 0.0 to go in the middle column, or column 63, and the x-value of +5.0 to go in the rightmost column, or column 125:

x-value	-5.0	0.0	+5.0
	↓	↓	↓
column	1	63	125

Taking any two of these relational pairs, we can determine the equation to convert real x-values into integer column numbers:

$$KOL = 63 + X \cdot 12.4$$

However, since this will always truncate a scaled x-value to the integer-valued column, we give the x-value a little “boost” so that the scaled value will

be rounded up (to the next column) or down, as appropriate. Also, since we have a symmetric circle, but only positive x-value roots calculated from solving the equation and using the square root, we actually have *two* x-values corresponding to each y-value (except for the top and bottom points of the ellipse). Thus we calculate a scaled x-distance from the middle column (63), *add* the scaled value to 63 for the positive x root, and *subtract* it for the negative x root point. The equation for the scaled x distance thus becomes:

$$M = \text{IROUND} ( X * 12.4 )$$

which rounds the distance up or down to the nearest integer value. This M value is then added to 63 for one point, subtracted from 63 to determine the other point.

Once the table of x- and y-values is set up, and the scaling determined, all that remains is to determine labels for both axes, and then begin filling the plot lines and printing them. The labels for the y-values are computed as we go along. We simply print the y-value on the line it represents. The x labels are a bit more complicated. We have 10 units of x-values spread over about 125 columns. To label the x-values in steps of 0.5 would involve 21 scale points, including both endpoints. If we print each of these in a 6-column field, that will just about match our 125 available columns. Thus, we set up an implied list to print the values from -5. to +5.0. in steps of 0.5, across the top of the graph.

Each plot line (the array LINE) is initially filled with blanks, and then a special character for the y-axis (use an ! or a | or whatever available symbol seems most appropriate). Then, the columns for the x-values of the points, given the y-value, are calculated, and a plot symbol (we used '\*') stored in those positions of the LINE array. Next, the y-value and array are printed. The top 25 y-values are done in a loop, then the 26th y-value ( $y=0$ ) is handled as a special case, to display the x-axis. This is done by filling the array LINE with axis symbols (such as +'s, -'s, or the like), except for the middle column, which still gets a '!; for the y-axis. Then, the negative y-values are printed in another loop.

The program which plots the circle is given here.

```
***** PLOT PROGRAM FOR CIRCLE *****
CHARACTER LINE(125)
REAL XLABEL, X, Y
INTEGER IROUND, M
IROUND (X) = X + 0.5
***** PRINT PAGE HEADER *****
PRINT 30, (XLABEL, XLABEL = -5.0, 5.0, 0.5)
30 FORMAT('1',50X,'PLOT OF CIRCLE X**2 + Y**2 = 25'/63X,
& 'X AXIS'/1X,'YAXIS, 21F6.1)
```

```

***** UPPER HALF OF GRAPH *****
Y = 5.0
DO 40 J = 1, 25
    DO 20 L = 1, 125
20        LINE(L) = ' '
        LINE(63) = '|'
***** CALCULATE SCALE X DISTANCE *****
M = IROUND ( 12.4 * SQRT (25.0 - Y**2) )
LINE(63 + M) = '**'
LINE(63 - M) = '**'
PRINT 22, Y, LINE
Y = Y - 0.2
22      FORMAT(' ', 1X, F4.1, 2X, 125A1/70X, '|')
40  CONTINUE
***** DO THE X-AXIS AS A SPECIAL CASE *****
DO 45 L = 1, 101
45        LINE(L) = '_'
        LINE(63) = '|'
        LINE(1) = '*'
        LINE(125) = '*'
PRINT 25, Y, LINE
Y = -0.2
***** NOW BOTTOM HALF OF GRAPH *****
DO 60 J = 27, 51
    DO 55 L = 1, 125
55        LINE(L) = ' '
        LINE(63) = '|'
        M = IROUND ( 12.4 * SQRT (25.0 - Y**2) )
        LINE(63 + M) = '**'
        LINE(63 - M) = '**'
        PRINT 22, Y, LINE
        Y = Y - 0.2
60  CONTINUE
STOP
END

```

Notice that, even though we only printed graph lines every other row, we did continue the y-axis mark on every line, to make it look continuous, by using the / to print two lines with FORMAT 22, the second of which only contains an axis mark.

You will find that each plot you do will present new problems, but you will have learned much that is useful from the previous plots. We constructed this plot one line at a time, working from the top, and calculating points in the plot

as we went along. We could have stored the entire picture of the circle in a two-dimensional array, but that would have been much less efficient in time and space utilization. However, there may be occasions in which we want to construct the *whole picture* in one development stage, and then print it out line by line. This is more closely analogous to the way a human being does a plot.

The human puts in points in any order that seems convenient, and is not constrained by beginning at the top and working down. This might be the case when data points are coming in some order in real time, and you want to store them in the plot array as they are received, without wanting to have to take time to sort them into descending y-order, for example. In such a case, we can use a two-dimensional array, of the appropriate number of rows and columns, and just fill it in any order desirable, printing out after it is completely filled. Before storing data points in the array, it would be initially filled with blanks, and any axes stored in the appropriate row and column. Otherwise, the same considerations of scaling and labelling still pertain to such a plot. Thus, you would have to know the *range* of values to expect before you could dimension the array.

For example, if you were going to plot points for a "scattergram" of points defined by two parameters—such as length and weight—you would have to determine the range of lengths and weights you expected as input, then figure out how many rows (say, for length) you would have for your y-axis, and how many columns (for weight) for your x-axis. These would determine the bounds of your array. The length and weight would then be read in and scaled appropriately for row and column position, and stored in their place in the array. The scattergram was examined in detail in problem 20 at the end of the previous chapter.

Imagine, for example, that the lengths you expect are in the range from 50 to 150 centimeters, and the weights will lie between 0.2 and 2.0 grams. You could have 101 rows in your array for the lengths, and if you are going to spread the picture over 121 columns, you could scale the weights to the correct column.

```

CHARACTER PIC(101,121)
INTEGER IROW, KOL
REAL LENGTH, WEIGHT, CONST
DATA PIC / 12221 * ' '
IROW (LENGTH) = IROUND(LENGTH - 49.)
KOL (WEIGHT) = IROUND (CONST*(WEIGHT - .2)) + 1
CONST = 120./1.8
...
READ*, LENGTH, WEIGHT
PIC(IROW(LENGTH), KOL(WEIGHT)) = '**'
etc.

```

When determining the scaling for a Cartesian plot, where the scales must be the same on both axes, first determine the xrange and the yrange. Then,

begin with one or the other, to decide how it will be spread over the picture area, and thus what the size of an x-unit (or a y-unit) will be. For example, if we decide to begin with the x-axis, determine over how many columns you will spread the x values (*ncol*). The size of an x-unit is then:

$$\begin{array}{ll} \text{xunit} = \text{ncol}/\text{xrange} & \{\text{in columns}\} \\ \text{or} \quad \text{xunit} = \text{ncol}/(10*\text{xrange}) & \{\text{in inches}\} \end{array}$$

This then determines the magnitude of the x-unit, which must be the same as that of the y-unit. Thus, the total size of the y part of the picture must be  $\text{yrange} * (\text{xunit})$ . This distance (in inches) multiplied by 8 (or 6) rows per inch gives the number of rows over which you should spread your y-values. From this, you can determine the appropriate step size for the y-values.

Analogously, you could begin by deciding how many rows (and thus how many inches) you intend to spread the y-axis values over; from this you can determine the size of a y-unit (in inches), which must be equal to an x-unit. The number of columns you will need then for the x-axis will be:  $(\text{yunit}) * \text{xrange} * 10$ .

## SCIENTIFIC VISUALIZATION

A very current topic in the scientific community is that of *scientific visualization*. NSF sponsored a workshop/symposium in February 1987 to discuss visualization in scientific computing, which had a record attendance from the various government research agencies. The concern of those attending the symposium, and of scientists in general, is that of how to handle the tremendous data "explosion" of our times adequately. Data can be collected in such great quantities in many scientific areas that there seems to be little hope of processing and interpreting it all. On top of this, computer simulations created by the scientists themselves can pour out practically unlimited data that they have generated, and which must be understood. (We will discuss simulations and models in the next chapter.)

Seismic data (of great interest for anticipating earthquakes), remote sensing and reconnaissance data, data from medical measurements, meteorological data, all are available in astounding quantity, begging to be analyzed. Many of the statistical "number-crunching" data reduction techniques simply are inadequate to represent the implications of such data, and important aspects may be overlooked altogether. Thus there is an increasing emphasis on creating visual representations of the data (using graphics techniques), to take a more direct channel to the human brain.

Data that has already been collected can be synthesized in visual form, or information currently being generated (by, say, a computer simulation

program) can be displayed and, on the basis of the insight gained from the graphics, can be modified (various parameters of the simulation can be tinkered with, and the result of these changes can be seen immediately in visual form). Thus, scientists are seeking new and more creative ways to use graphics to display their data and analyze their problems.



## FRACTALS

A fascinating area of graphics is that of *fractals*, a term coined by Benoit Mandelbrot. A Euclidean geometry of straight lines and regular curves (such as circles) is inadequate to describe most of nature. It cannot describe mountains, clouds, coastlines, and many other natural shapes. A new geometry is needed to describe the irregular, fragmented shapes of nature—a family of shapes called *fractal sets* by Mandelbrot, from the Latin “*fractus*,” which means irregular or fragmented (from the root “to break”). Such shapes are “self-similar” (that is, they look the same independent of scale—whether magnified or at a distance) and self-replicating; they are recursive patterns. James Gleick writes (*Chaos*, p. 98), “In the mind’s eye, a fractal is a way of seeing infinity.”

We are already familiar with *iterative* processes, such as those we looked at in Chapter 9—functions to approximate infinite series, square roots, and so on. In these iterative processes, each new value comes from evaluating the same formula with the previous value plugged in. Thus, every new approximation ( $X'$ ) for the square root of a number  $A$  came from evaluating the following equation for the previous value ( $X$ ):

$$X' = (X + A/X)/2$$

Similarly, population growth or compound interest can be calculated for each new time period that has passed as the previous population (or principal amount),  $P$ , times (one plus the rate of increase,  $R$ , for that time period):

$$P' = P(1 + R)$$

The “orbit” of a particular value,  $x$ , for a certain formulaic transformation (such as the square root of  $x$ , or  $ax + b$ ), is the set of successive values it takes on, if the formula is iteratively applied to the result of the last transformation.

If such iterative transformations are applied to complex variables, instead of to real variables, we have the basis for the fractal sets discussed. They involve examination of quadratic functions in the complex plane of the form:

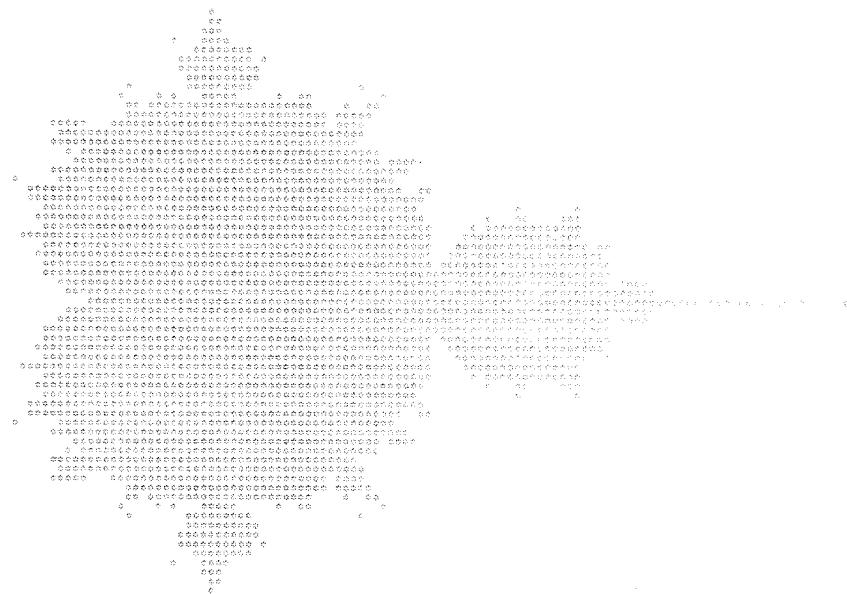
$$z^2 + c$$

where  $z$  and  $c$  are complex; this results in a variety of different behaviors, depending on the choice of value for  $c$ . We will suggest several exercises at the end of the chapter to create different *Julia sets* (named after the French

mathematician Gaston Julia, who studied them during World War I) that depend on the choice of value for  $c$ . The *Mandelbrot set* is a sort of "catalogue" of all the possible Julia sets, or fractal shapes constructible with this quadratic function. It is a set, or collection, of points  $c$  in the complex plane, such that if you square the value and add  $c$ , and then square this result and add  $c$ , and so on, the sequence of values does not "escape" to infinity. Thus, any value  $c$  for which a certain number of iterations (let us say 30) can be performed without the value of the result exceeding 2, will be a member of the Mandelbrot set.

Our knowledge of complex numbers and of plotting can be combined to create a program that will plot a version of the Mandelbrot set. This program was contributed by our student and friend, Charles H. Recchia.

```
*****
*          MANDELBROT SET          *
*****
*****      WRITTEN BY CHARLES H. RECCHIA      *****
*****      CALCULATE THE ORBIT OF A COMPLEX FUNCTION  *****
      CHARACTER LINE(100)
      COMPLEX A, K, Z, FXN
      FXN(Z) = Z**2 + K
*****      PROGRAM WILL PLOT 80 LINES (ROWS)      *****
      DO 40 Y = 1.2, -1.2, (-2.4/80)
      DO 10 L = 1, 130
10          LINE(L) = ' '
      M = 1
*****      PROGRAM WILL PLOT 130 COLUMNS      *****
      DO 30 X = -2.0, 0.8, (2.8/130)
      K = CMPLX(X,Y)
      A = CMPLX(0.0, 0.0)
      I = 1
20          CONTINUE
      A = FXN(A)
      I = I + 1
      IF (CABS(A).LT.2 .AND. I.LT.30) GO TO 20
*****      TEST WHETHER THE POINT HAS ESCAPED      *****
      IF (CABS(A).LT.2) THEN
          LINE(M) = '**'
      END IF
      M = M + 1
30          CONTINUE
      PRINT 35, LINE
35          FORMAT(' ', 130A1)
40          CONTINUE
      END
```



Note that each of the 130 points  $(x,y)$  tested for a given row  $(y)$  will either satisfy the conditions of the set, and so be included (stored in LINE), or not (they "escape"). Thus, a counter M which represents the position of the point in LINE is just increased each time, and a point stored or not, as appropriate. This is a clever device used by the programmer that avoids the need for any scaling to determine where in LINE to store the point.

The output from the program (somewhat scaled down) is shown above.

It is also interesting to note that special-effect graphics based on fractal geometry were used in the movie *Star Trek II: The Wrath of Khan* to represent the effects of Project Genesis in creating new life on a dead planet, as well as in several other big science-fiction movies.



## SCREEN GRAPHICS

There are many different graphics packages that are available, which will allow you to use devices with much greater resolution than we have had on the printer or screen in these examples. They will allow you, for example, to access each *pixel* on a high-resolution graphics screen, where we have only been able to access a *character* location on the screen, which is a rectangle or perhaps  $7 \times 8$  pixels or larger. Similarly, there are other kinds of high-resolution displays, including plotters, available for use with the right graphics packages. If you are interested

in pursuing these specialized applications in conjunction with FORTRAN, we refer you to:

Ian O. Angell and Gareth Griffith, *High-Resolution Computer Graphics Using FORTRAN 77* (Halsted Press, 1987).

You may also find ways to feed files created by your FORTRAN programs in as data to these special packages. We recommend that you look into the devices and packages available to you at your system location.



## SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

The printer or screen can be used for plotting data and creating pictures. Generally this is done using characters, either by tabbing to calculated locations and placing a symbol, or with character arrays (for histograms, filled with the appropriate number of characters, or for more general plotting, an initially blank 1-D or 2-D character array into which symbols are placed according to the results of calculations). Plots must be scaled to look reasonable on the output device, so that the writer of the program should be aware that the printer generally has 132 columns, 10 columns per inch, and either 6 or 8 rows per inch (down an 11-inch page), and the screen usually has 8 columns per inch and 4 rows per inch. These proportions should be taken into account, especially when plotting graphs where the axes should be commensurate (as in Cartesian graphs).

An x-coordinate can be scaled to its corresponding place (M) in a 1-D character array (or in a row of a 2-D array) by determining an appropriate multiplier and additive factor:

$$M = A \cdot X + B$$

Several plotting examples requiring scaling are discussed at length in the chapter; the reader is referred to these.

"Scientific visualization" is a currently "hot" topic in the scientific community, since the current data explosion is beginning to exceed the capacity of scientists to handle, without significantly *reducing* the data, e.g., by picturization.

The topic of *fractals* was discussed briefly in the chapter.



## EXERCISES

1. Write a program which will read in a large amount of data from a weather-monitoring station regarding humidity readings, and create a histogram (it

should be fairly fine-grained, since humidity values do not vary widely) of the occurrences of humidity in different subranges of the range.

- ◆◆◆◆◆ 2. Write a program to plot your biorhythms for the next 100 days, as discussed in the text. (This is the program that you have been building up to in stages through the text exercises.)

3. A “density plot” allows you to get a three-dimensional effect with a two-dimensional graph. A function of *two* variables,  $f(x,y)$ , will give you values to be represented in a *third* (*z*-) dimension. To accomplish this, you can use characters of different darkness (density) levels to put into each  $(x,y)$  location, to reflect the magnitude of the  $f(x,y)$  value. A range of ten characters of increasing density you can use is:

. , - = + X & \$ @ #

You can store these characters in a 10-element array, and then pull out the appropriate one for the *z*-value in the plot.

For example, the function:

$$f(x,y) = x^2 + y^{1/2}$$

describes a three-dimensional “surface.” Write a program which will create a density plot of this function, for values of *x* and *y* that both range from 1 to 20. Calculate what the lowest *z*-value you can expect is (2), and the largest ( $20^2 + 20^{1/2} = 424.47$ ), divide this range into 10 segments, and plot the appropriate density symbol for each *z*-value.

*Note:* A more effective density plot might be achieved by using the method of overstriking we discussed.

- 4. Plot the ellipse whose equation is:

$$x^2/25 + y^2/36 = 1$$

Write one version of the program to run to the screen, and a larger-scaled version to run to the printer. This should not be a great deal different than the circle program in the text.

- 5. Plot a graph that shows the intersection of a parabola and a straight line. The equation of the parabola is:

$$y^2 = 6x$$

and the equation of the straight line is:  $2x = y + 6$ . By examining a rough sketch of the two equations, we discover that the parabola only has positive *x*-coordinates (or *x*=0), and that the parabola and the straight line intersect at the point (6,6). Both the parabola and the straight line are infinite, not nicely limited as was our ellipse. Since we obviously cannot plot an infinite graph, we must limit our ambitions to something reasonable.

The positive x,y quadrant shows us two curves which intersect at the point (6,6), and that the parabola touches the origin at the time that the straight line crosses the x-axis at x=3. We thus see that the parabola is symmetric about the x-axis and we determine that the two curves are most interesting, for our purposes, in the range of positive values up to the point at which they intersect. To display the parabola adequately, we also want to plot its mirror image below the x-axis. If we then plot all the x-values from 0 to 6, and all of the y-values from 6 to -6, we will see the symmetry of the parabola, the points where it crosses the line, and we can do all of this without involving any of the negative part of the y half of the plane. The lowest value of y we will include (-6), is the point at which the straight line crosses the y-axis. Thus the range of values we want to plot is determined, and we need only determine an appropriate scale.

6. The equations to "rotate" a visual object through an angle A, about pivot point (c,d) are:

$$x' = (x - c) \cos A - (y - d) \sin A + c$$

$$y' = (y - d) \cos A + (x - c) \sin A + d$$

Create a nonsymmetric visual object for your plotter output, either by using equations or by reading in the coordinates of its outline to your program. Have the original object outlined using some dark character such as the number sign '#'. Now perform *two* different rotations of the object about a specific pivot point, using two different characters (say, '\*' and '+') for the two rotations, and output the entire picture (original plus the two rotations) to the printer.

7. Rotations, as in the previous problem, can be determined by the use of matrices. If a point  $(x_0, y_0, z_0)$  is rotated through an angle A about the z-axis, the resulting coordinates  $(x, y, z)$  are determined by multiplying the original coordinates by the matrix:

$$\begin{bmatrix} \cos A & \sin A & 0 \\ -\sin A & \cos A & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

A rotation of the same point by an angle B about the x-axis can be represented as a multiplication by the matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos B & \sin B \\ 0 & -\sin B & \cos B \end{bmatrix}$$

Use this information to write a program which will read in the coordinate positions of a rectangle's corners in the x,y-plane, create a picture of the

rectangle, showing its boundaries (use a '\*' character for the original rectangle boundaries), and then superimpose on this picture the same rectangle after it has been rotated by a specified angle (say,  $30^\circ$ ) about the x-axis; use a different symbol (say, '+') to represent the boundaries of the rotated rectangle. After the "picture" (that is, the array), containing both rectangles has been completed, print it out.

- 8. Write a program to plot a sine curve, going down the page (say, for three cycles), analogous to that in the biorhythm problem (2), but this time *fill in* the points between the axis and the curve.
- 9. Plot the trajectories for a projectile fired at different angles (say,  $30^\circ$  to  $60^\circ$  in steps of  $5^\circ$ ), all on the same graph. The acceleration of gravity  $g = 32.2$  feet/sec $^2$ .
- 10. A bomber releases a bomb from a height of 500 feet while the plane is flying 300 miles per hour. Plot the trajectory of the bomb as it falls, until it hits the ground. Also print out the horizontal distance it travelled after it was released.
- 11. If a sphere rolls along a flat surface, a point of the sphere will trace out a curve called a *cycloid*. If we look at a two-dimensional version of the sphere (that is, a circle), and pick a point P on the circle (initially at the bottom), we can plot the curve that point P traces out as the circle rolls along the floor at a constant velocity  $v$ . If the radius of the circle is  $R$ , then its angular velocity  $A = v/R$ , and the equations of motion of the point P are represented by:

$$x = R(At - \sin At)$$

$$y = R(1 - \cos At)$$

if the point P begins at the origin (0,0) at time  $t = 0$ . Plot the trip of point P for at least 5 revolutions of the circle.

- 12. A *square wave* and a *saw-tooth* wave can be approximated by the superposition of a number of sine waves (if an infinite number of sine waves were combined, the synthesis would be perfect). Using the following infinite series representations of square and saw-tooth waves, plot approximations to both, varying the number of terms you include in the series. Arrive at an empirical evaluation of how many terms are needed to make the approximations "good enough." Begin by adding at least 2 terms.

$$\text{Square wave: } \frac{4}{\pi} (\sin \omega t + \frac{1}{3} \sin 3\omega t + \frac{1}{5} \sin 5\omega t + \dots)$$

$$\text{Saw-tooth wave: } \frac{2}{\pi} (\sin \omega t - \frac{1}{2} \sin 2\omega t + \frac{1}{3} \sin 3\omega t + \dots)$$

- 13.** *Radioactive decay.* A radioactive substance decays according to the formula:

$$M = m e^{-kt}$$

where  $m$  is its original mass,  $M$  its mass at time  $t$ , and  $k$  is a constant representing its rate of decay. The *half-life* of a radioactive substance is the time required for its original mass  $m$  to be reduced to half ( $m/2$ ). Thus the half-life  $h = \ln(2)/k$ , or  $k$  can be expressed in terms of the half-life ( $k = \ln(2)/h$ ).

If you know that the half-life of Thorium 234 ( $\text{Th}^{234}$ ) is 24.1 days, plot the amount of Thorium, beginning with 100 units, over a period of 120 days.

**14.** Two planes are travelling toward one another. The first begins at point  $(x_1, y_1, z_1)$  at a velocity  $v_1$  with components  $(vx_1, vy_1, vz_1)$ , and the second begins at point  $(x_2, y_2, z_2)$  with a velocity  $v_2$ . Plot the projection of the courses of the two planes onto the  $x, y$ -plane until either they collide or until they have safely passed one another. Assume that one minute (60 seconds) will be enough time in which to resolve the question of whether they will crash or pass safely. Since they are not actually point masses, assume a crash will occur if the two planes, considered as point masses for purposes of the plot of their trajectories, come within 30 feet of one another.

- **15.** Write a subroutine which will accept a clock time in hours and minutes and draw a clock face that will represent the time in analog form (that is, not a digital clock or watch).

**16.** Add a second hand representation to the problem in problem 15.

**17.** A "window" in graphics restricts what can be "seen" to what will fit within the boundaries of the window. Write a subroutine which will accept the coordinates of the vertices of a window, and the parameters for the equations of three lines (of the form  $y = a_i x + b_i$ ). Have your subroutine print out the boundaries of the window, and *only* the segments of the three lines which fall within the window.

**18.** An eclipse of the sun occurs when the moon passes between the earth and the sun. If we assume that, from the perspective of the earth, the moon and the sun *appear* to be circles of the same radius, draw a plot representing the passage of the moon across the face of the sun. First, create a picture in which the outline of the sun remains fixed in the center of the page, and draw successive pictures of the outline of the moon (using a different character) as it passes across the face of the sun. Begin with the two circles adjacent (touching), and end when the moon has completely passed the sun. Divide the time of passage into 21 segments (that is, have 21 successive versions of the moon's position).

Now modify the program described so that you display, in sequence, 21 different versions of the "light" available from the sun. "Fill in" the portion of

the sun which is not shielded by the moon with a character (such as '\*'s); also indicate the complete outline of the moon on your picture. Print out next to the picture what percentage of the sun's light is not shielded.

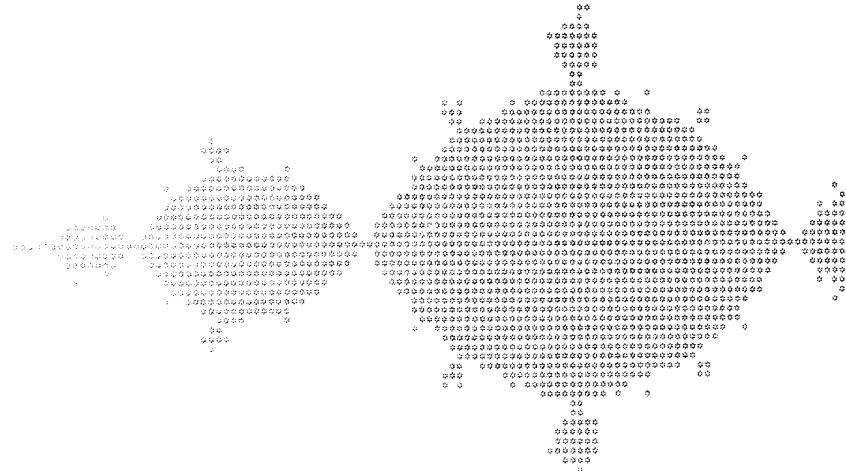
**19.** Modify the Mandelbrot set program in the text so that it allows the user to input the coordinates of two points ( $(x_1, y_1), (x_2, y_2)$ ) that represent the coordinates of, respectively, the upper left corner and the lower right corner of a "window" on the Mandelbrot set, and have the program print out only the points of the set that fall within the window. This can then be used to input the coordinates of four different quadrants of the set, and run the program four different times to create plots of these quadrants. The plots can then be pasted together to create an enlarged, more detailed, version of the Mandelbrot set.

**20.** The program in the text which creates a plot of the Mandelbrot set can be modified to plot a number of other interesting sets called "Julia sets." The main difference is that, for each of these sets, a value for the complex constant  $c$  (or  $K$ , as it is called in the program), will be input, so that it can be varied to create different Julia sets. This complex value will be input for  $K$  before the loops in the program, and the value of  $A$  will be the complex point defined by  $X$  and  $Y$ , rather than starting at zero. Otherwise the program will operate much the same way, and the range values used for the Mandelbrot set will work well for the Julia sets also.

Once the program is written, input different values for the complex constant  $K$ . We suggest the following values, which should create the Julia sets pictured:

- |                 |                 |
|-----------------|-----------------|
| (a) (-1.0, 0.0) | (b) (0.3, -0.4) |
| (c) (.36, .1)   | (d) (-0.1, 0.8) |

(a)



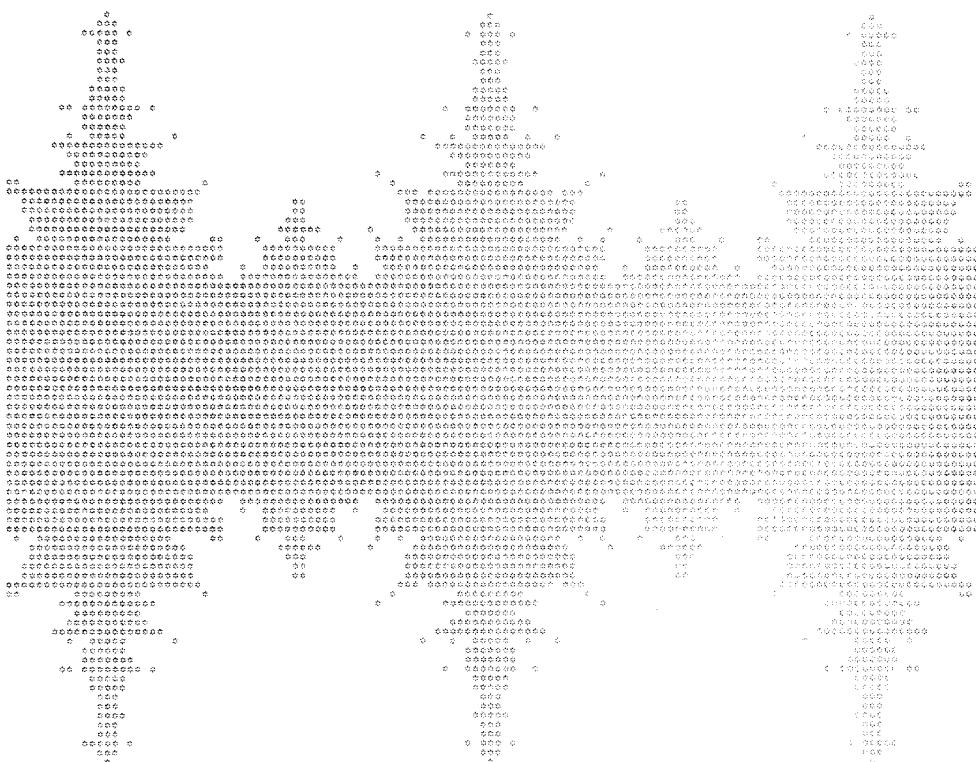
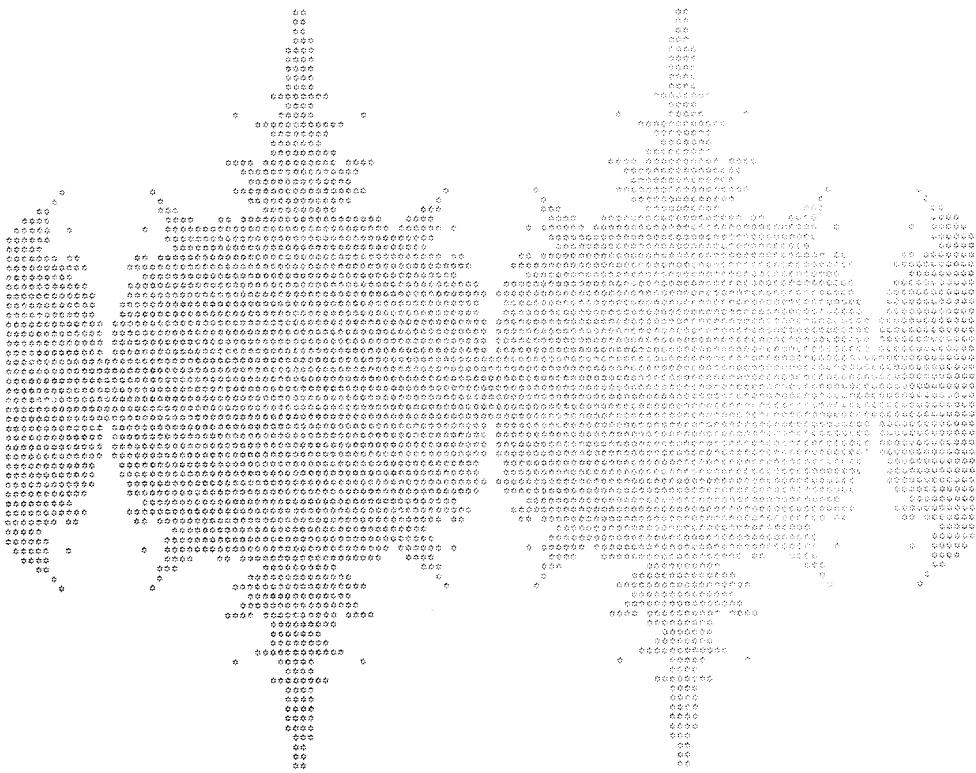


(b)

(c)



(d)

**COS Z****sin z**

- 21.** The program developed in problem 20 can be modified to produce other interesting Julia sets, by taking the sine or the cosine of a complex value, instead of using the quadratic function  $z^2 + c$ . You will have to play with the limits of  $x$  and  $y$  for these programs, and the value which constitutes "escape" to infinity, but with good values you should get plots like those on the preceding pages.
- **22.** Using an array, the random number generator, '+' carriage control, and a delay loop, write a program that will simulate a slot machine with three slots in which at least eight symbols (for example, '+++', '\*\*\*', '% % %', '\$\$\$', '@@@', etc.) may appear (randomly). Have the slot machine run through at least 20 "spins" per game, stopping if there is a "winner" (all symbols match), and printing out a message to that effect. This will create interesting effects to the screen, but do not try to run it to the printer (there are many overstrikes). Allow the player to enter a different seed each time the program begins, and allow choice of whether to continue playing after a game.
  - 23.** Using some form of variable formatting (discussed in the previous chapter), print out Pascal's triangle (the table of binomial coefficients, or combinations, we worked on earlier) so that it really looks like a triangle; that is,

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
etc.

Have the program accept a number of lines (up to 20) to be generated in the triangle.



## SUGGESTED READINGS

Barnsley, Michael. *Fractals Everywhere*. Boston: Academic Press, 1988.

Devaney, Robert L. *Chaos, Fractals, and Dynamics: Computer Experiments in Mathematics*. Reading, Mass.: Addison-Wesley, 1990.

Gleick, James. *Chaos: Making a New Science*. New York, N. Y.: Viking Penguin Inc., 1987.

- Mandelbrot, Benoit B. *The Fractal Geometry of Nature*. San Francisco: W. H. Freeman and Company, 1983.
- Mandelbrot, Benoit B. *Fractals: Form, Chance, and Dimension*. San Francisco: W. H. Freeman and Company, 1977.
- Nielson, Gregory M., and Bruce Shriver (eds.). *Visualization in Scientific Computing*. IEEE Computer Society Press, 1990.
- Peitgen, H. -O., and P. H. Richter. *The Beauty of Fractals: Images of Complex Dynamical Systems*. Berlin: Springer-Verlag, 1986.
- Rapaport, Dennis. "Visualizing Physics." *Computers in Physics* 3, no. 5 (Sep/Oct 1989), 19ff.
- "The Visualization Roundtable." In *Computers in Physics*, 2, no. 3 (May/June 1988), 16–26.
- Wolff, Robert S. "Visualization in the Eye of the Scientist." In *Computers in Physics*, 2, no. 3 (May/June 1988), 28–35.

# CHAPTER 13



## SOME INTERESTING APPLICATIONS

A variety of useful scientific problems involve modelling the behavior of a system, or simulating a process to predict its future. FORTRAN provides the capability to generate "random" values for cases involving probabilities. Interactive programs allow a program to react in real time to external inputs, such as in game-playing programs or ground control of satellites. Computer pattern recognition techniques have many applications, from check-reading to controlling a robot in a hostile environment. This chapter introduces these topics, as well as the creation of new data types and structures.

An animated, computer-generated image of a cockroach creeps uphill.

*"Thou cunning'st pattern of excelling nature."*

- William Shakespeare, Othello V, ii, 1

## SIMULATIONS AND MODELS

### Why Model?

Many problem situations may have a *random* element, or may involve elements that change deterministically over time in a manner so involved that a simple equation or two will not capture the complexity. In such situations, you will want to employ a *simulation*, or a model, of the situation, and then let your program examine how it will unfold over time.

A *deterministic* problem is one in which all of the varying parameters are known, but their interaction may be complex. An example would be a growth or decay problem with several factors interacting. For example, I get a 10% raise every year, beginning with a gross salary of \$24,000, but 30% is withheld (until my salary reaches \$32,000, at which point 35% is withheld) for taxes, Social Security, etc.; I always spend 10% of my net salary (after withholding) for fun; and I have fixed expenses (rent, food, etc.) of \$10,000 a year. I put anything that is left into the bank for savings. Given these parameters, how long will it take me to save \$50,000? \$100,000?

You could readily write a program that would model my income year by year, as well as the withholding and expenses, and determine the total accumulated in the bank, counting years until the desired total was reached. You have already worked on a few modelling problems like this one.

Another deterministic model is that envisioned by Laplace, of a universe which obeys deterministic, billiard-ball collision type laws. In his model, if you knew the positions of all of the particles in the universe and their motions (velocity—or acceleration—and direction) at any given point in time, you would be able to predict the status of the universe at any future time. Furthermore, you would be able to extrapolate backward and discover the state of the universe at any past time as well.

However, many problems you encounter will not have fixed parameters like this one, but will have a *probabilistic*, or random, element to them. We can also write programs to simulate the activity of such a system, but these programs will be somewhat more complicated and will involve using *random numbers*.

Simulations are done for many reasons. You want to predict the future, instead of simply waiting for it to happen. You may wish to model something (like a galaxy) that is too large to bring into your laboratory and see what happens as a result of certain possible changes, such as a star going nova; in such a case, you develop a *reduced-scale* model. If you wish to model something

which is too small to work on directly in your lab, such as an atom or a molecule, you may develop an expanded model to experiment with.

Design engineers work with scale models of automobiles, planes, bridges, buildings, and the like to determine their feasibility. A model may be a physical replica—usually in reduced scale—or it may be a *mathematical model*, with a system of equations representing the interrelationships in the system. Such a mathematical model can be implemented—and experimented with—on a computer, with numeric representations of stresses and strains, and of other parameters.

Many simulations involve probabilities, or a random element. If you are modelling a game of chance, such as the roll of a die, which face of the die turns up on a particular roll is unknown, though you know the relative probabilities of the six options. Since a simulation would have to work with a particular value for the roll, you need to simulate this random occurrence. Other models incorporate random elements because the assumption of randomness does not imply any particular structure when none is known. Thus, in modelling the brain, random connections are assumed, so as not to bias the model in some untoward way. In economic models, a random model does not imply a bias toward any particular theory, since none is more justified than any other.

You may want to model to predict the future, or to examine the reliability of a proposed theory, or simply not to damage the system being modelled by actual experimentation with it (as in the case of a brain). A model on the computer allows you to perform experiments which “look into the future,” or test the results of various “ tweaks” to the system, without actually changing anything in the real world. You want your model to be as realistic as possible, so you build into it as much as possible of what is known about the system you are modelling, to make yours a “likely story.” The more your model has in common with the reality it simulates, the more believable your results will be.

There has been considerable interest in simulating the future of this planet, beginning with the work of the Club of Rome in the 1970s which resulted in the publication, *The Limits to Growth*. The current state of various resources and their use, as well as population data and the current growth rate, were built into the model, and predictions made if the trends continued; the predicted results were catastrophic!

Models are also used frequently to simulate *queues*, or waiting lines, in an attempt to improve service. Thus, a model could simulate traffic lines at a particular intersection, or grocery checkout lines, or the like. The designer of the simulation should have available certain statistics regarding the sampled distribution of such traffic in the past. But a statistical distribution does not tell you how many cars, or shoppers, there will be at a particular time (and it is hardly ever the average). A good simulation, however, could pick a representative number from the distribution and use that number to stand for the value at the time in question. If this is done for a great many instances, and the results examined, the study should provide a realistic model of what might actually

happen in the future. Thus, reasonable recommendations could be made regarding the optimum way to set the traffic light to facilitate traffic flow, given the random variations to be expected, or the best way to use checkout people in the grocery store.

The movie "War Games" made such simulations well-known, but they have been carried on for years, primarily by the RAND Corporation, in work for the Department of Defense. Such models need to take account of randomly occurring events, and early in their work, RAND constructed a table of a million random digits to use in their situations. Such a table can be referenced (perhaps stored on disk) to get random values for a simulation, but it is not the most efficient way to proceed.

## Random Number Generators

The mathematician John von Neumann, who did early work with Oscar Morgenstern on economic theory and recognized its connection with game theory, proposed a simple method for generating "pseudo-random" numbers according to an algorithm. They must be called psuedo-random because truly random values are totally unpredictable, whereas anything that comes from an algorithm is predictable if you know the algorithm. However, if we can design an algorithm that creates a sequence of values where there is no discernible pattern (to anyone who does not know the key), and which gives values that are distributed as we would want our random values distributed, it should not matter that they are only *pseudo*-random—they will do as good a job as truly random values. As has been pointed out, the person using the random numbers should not be affected by their origin, "since the question he should be asking is not 'Where did these numbers come from?' but 'Are these numbers correctly distributed?', and this question is answered by statistical tests on the numbers themselves." (Hammersley and Handscomb, *Monte Carlo Methods*, p. 25.) If you cannot effectively tell any difference, then there *is* no important difference to the job to be done.

Von Neumann's early attempt at a simple random-number sequence generator was called the "mid-square method." Given an arbitrary starting point, the algorithm will generate a sequence of numbers uniformly distributed over a given range. Von Neumann suggested taking an n-digit number, squaring it (which gives a value  $2n$  digits long), and taking the middle n digits of the result as the next number in the sequence. Thus, each new number in the sequence comes from the previous one. This is a "recursive" definition. We must have a way of selecting a starting value, which is called the "seed" of the generator. To illustrate the mid-square method, let us look at a simple sequence of 2-digit "random" numbers using this technique.

We use a 2-digit seed for the sake of clarity here, but in reality if this method were used, one should select a value for n as large as the computer being used

will accommodate; the larger the values used, the better the randomness of the sequence. To begin the mid-square method, let us arbitrarily choose the prime number 11 as the initial value, or seed, of the generator. The next value comes from squaring 11 (to get 0121), and taking the middle two digits of the result (i.e., 12) as the next value in the sequence. We then square 12, getting 0144, and take 14 as the next random value, etc. The sequence proceeds as follows:

11, 121, 144, 196, 361, 1296, 841, 7056, 25, 004, 0, 0..

that is,

11, 12, 14, 19, 36, 29, 84, 5, 2, 0, 0, 0 . . .

We can readily see, as von Neumann recognized, that the terms in this generator algorithm tend to degenerate to zero (and once a zero value is hit, it remains stuck on zero). Another weakness of the algorithm is that it tends to "cycle," or repeat, too frequently in many cases. Clearly, for an n-digit size sequence, the longest possible cycle would be  $10^n$ ; but in most cases, it is much shorter than that. Consider, for example,

6100, 2100, 4100, 8100, 6100, . . .

In a test run at the National Bureau of Standards, sixteen sequences of 4-digit numbers were generated. Twelve of them ended in the 6100 sequence above; two ended in 9600, 1600, 5600, 3600, 9600; and two degenerated to zero. The mid-square method is thus seen not to be optimum, but it does give us the general principles of a random number generator. It also makes a nice programming exercise for the reader.

Incidentally, we should mention at this point that almost every FORTRAN compiler contains a random number generator; as a result, you probably won't have to write your own. Just in case you are in a situation where you have to write one—because either none is available or the one on the system does not have good statistical characteristics—you should know how. Furthermore, understanding these techniques of generation makes you a more knowledgeable user of any random number generator.

Probably the most common random number generator algorithm is the mixed linear congruential method. In this method, the calculation of each new random number is, as indicated, based on the previous one in the sequence:

$$X_n = (A * X_{n-1} + C) \text{ modulo } m$$

To choose a simple example, for clarity, let us take  $A = 3$ ,  $C = 7$ ,  $m = 100$ , and a seed value  $X_0$  of 5; we then get:

5, 22, 73, 226, 85, 262, 193, 286, 265, 202, 13, 46, 145, . . .

which gives a nice appearance of randomness. It is advisable to choose A and C relatively prime to each other.

Notice that these generators have created sequences in a particular range, determined by the choice of modulus  $m$  (in which case all random values  $x$  generated lie in the range  $0 \leq x < m$ ), or by the number of digits  $n$  in the numbers (in which case they lie in the range  $0 \leq x < 10^n$ ). Thus, to use one of these generators, one would have to know the value of  $m$  (or  $n$ ) in order to know what range the values would fall into. This would be a great inconvenience as one moved from one machine to another with a program that required a random number generator. Thus the values from a random number generator are usually *normalized* to lie in the range from 0 to 1 (this is done by dividing the values by the modulus  $m$ , or by  $10^n$  in the algorithms which generate  $n$ -digit numbers). In the examples we used earlier, this would merely divide all of the numbers through by 100, creating fractions, since we either generated 2-digit numbers or numbers modulo 100. Note also that the normalized random numbers generated are always less than 1.

## Scaling the Generator to Actual Distributions

All of the random number generators we have discussed so far will give you a roughly *uniform* distribution of values, normalized to lie between 0 and 1. Yet, more often than not, the range 0–1 may not be the range of random values you are looking to simulate. If you are generating random points in a unit square, which we will do in the Monte Carlo section, the distribution is fine as it stands. However, it is more likely that you will be dealing with a problem that needs randomly distributed real values in the range from  $a$  to  $b$ , or integer values from  $m$  to  $n$ . We thus need to look at the 0–1 distribution from the random number generator, and determine how to modify it to fit the distributions we need.

It is very likely that your FORTRAN system will have a random number generator in it for you to use, so you will not have to write your own, unless you think you can improve on the characteristics of the one available. Unfortunately, however, these generators do not fall under the 1978 ANSI Standard, or any earlier Standard, so there will be a variety in the random number generators you may encounter. Some of those commonly found are described here. In general, the generator is a system function, and might be named RANF, RANDOM, RANDU, RAN, RND, or something else. It takes one argument, which may be a dummy, but more often is used to represent the “seed” of the generator. If you do not provide an initial seed, either a default value will be provided by the function or it will use a seed of zero. If you do not provide a different seed, every time you run a program using a particular generator, you will get the *same* sequence of random numbers. In some instances, this might be desirable, for instance, if you want to repeat a simulation experiment with the same “random” values, but varying one or more parameters of the system; however, more often you want to “mix things up” on different runs.

For example, some systems use the random generator function RAN(N), or RANDOM(N), which behaves in the same way. If you do not specify N initially, it will probably begin the sequence with a seed of 0. However, if you give the variable N (or any other integer variable you choose to use in its place as argument to the function) a value at the beginning of the program, it will then generate a sequence of random numbers based on the initial value of N. You have seen, in the algorithms for the mid-square generator and the linear congruential method, how the *seed* value affects the sequence; a different seed creates a different sequence of "random" values.

You may want to ask the user to choose and enter a large random integer to seed the random sequence. As the calls to the function proceed in the program, the value of your variable N will be changed by the generator function. For example, the following program generates 1000 random numbers on a base, or seed, of N = 99931.

```

N = 99931           {could be read in instead}
DO 88 I = 1, 1000
  X = RAN(N)
  .....
88 CONTINUE          {simulation business}

```

The random number generator RANF(N), as used for example on CDC machines, generates random values over the range 0–1, but the argument N is a dummy argument. In order to initialize a different seed for this generator, the user must call a related subroutine, CALL RANSET(nn), where nn is a value to seed the generator. To determine the current seed of the generator, use CALL RANGET(ns), where ns receives the current integer seed.

A function RAND(RMAX) has been used, which generates random numbers in the range 0–RMAX (so a call to RAND(1.0) would generate a number in our more standard base range of 0–1). A generator might be called RAN, or RND (as it is in BASIC), or RANDU, or something else. You should check in the FORTRAN manual for the system you are using to get the details of the random generator available—its calling sequence, its name, and how you can alter the seed. Of course, if none is available, by now you are well prepared to write one of your own. In our examples in this book, we will use the RAN generator first described.

*Note:* Fortran 90 will have a random number subroutine, RANDOM\_NUMBER, which returns a value or an array of random numbers in the range 0 to 1. The subroutine RANDOM\_SEED can be used to initialize the pseudorandom sequence.

Suppose you need to generate real values in the range from 0 to RMAX, and you only have a generator which gives you values from 0 to 1. Then you simply scale up the values from the generator by multiplying them by RMAX:

```
***** GENERATE A RANDOM REAL BETWEEN 0 AND RMAX *****
Y = RMAX*RAN(N)
```

What if you wanted to generate real values in the range from a to b, for example, in the case where you want to randomly generate x-coordinates (or other values) between a and b? To get a real value x in the range  $a \leq x < b$ , we simply have to scale and shift the value q from our generator, which is in the range  $0 \leq q < 1$ . To do this, we multiply by the magnitude of the range of values we want to create ( $b-a$ ), and add the lower limit (a) of the range, thus:

```
***** GENERATE A RANDOM REAL BETWEEN A AND B *****
X = (B - A)*RAN(N) + A
```

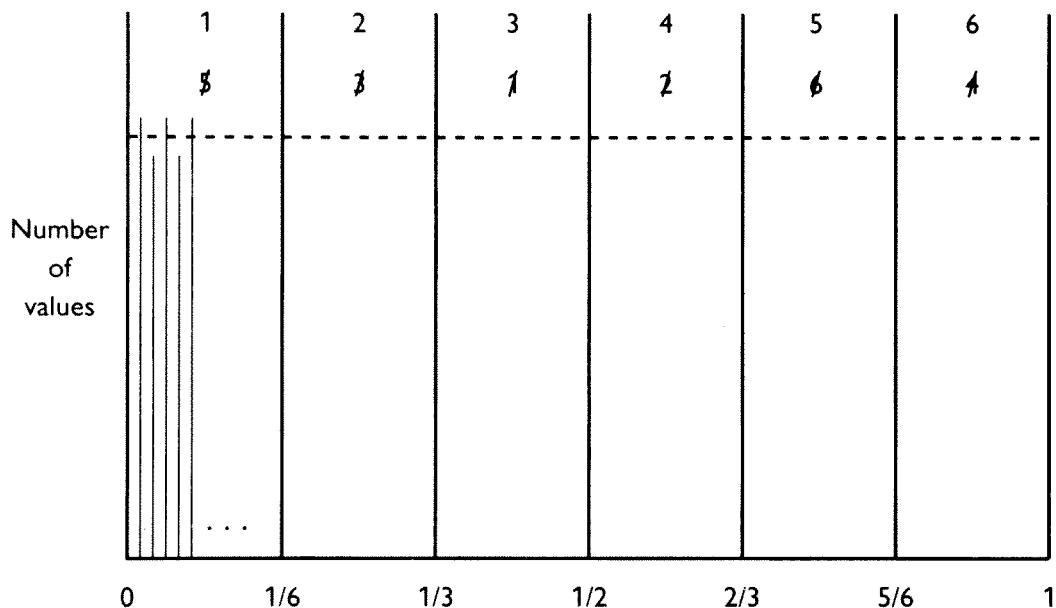
What if we need random integer values in a certain range for our simulation, such as the integers 1, 2, 3, 4, 5, or 6, uniformly distributed, to simulate rolling an unloaded die? To do this, we must look carefully at the distribution of real values our random generator gives us. If we were to generate thousands of values from our generator, and plot a histogram of the distribution, by counting how many values fell between 0 and 0.01, between 0.01 and 0.02, etc., our histogram should look roughly flat (if it is a good generator). That is, roughly the same number of values from the generator should fall into each equal-sized subrange we created.

Now, if we were to divide this distribution into as many equal-sized parts as the number of integers we want to simulate, then we could assign one integer to each subrange we created, and it would have the same likelihood as any other integer representing a different subrange. In the case of our die, where we want an integer equally likely between 1 and 6, we could divide the distribution into 6 equal subdivisions, and arbitrarily assign each one to a different integer in the range (as shown on the diagram: for instance, if the random value x generated fell in the range  $0 \leq x < 1/6$ , that would be called rolling a 5; if it fell in the range  $1/6 \leq x < 1/3$ , it would be called rolling a 3, etc.). Then, when we produced a real number from the generator, we would just test it to see which range it fell into, and then assign an appropriate integer.

```
X=RAN(N)
IF (X.GE.0 .AND. X.LT.1.0/6.0) THEN
    NROLL = 5
ELSEIF (X.LT.1.0/3.0) THEN
    NROLL = 3
etc.
```

This is not too bad for integers from 1 to 6, since it only requires six IF tests. However, if we had 600 or 6000 different integers to generate, this would not be a good approach. Notice that the choice of integer assigned to each

subdivision of the distribution was arbitrary, and if we choose in a more orderly fashion—the smallest integer to the leftmost subrange, and so on—we will find that there is an easier way to make the transformation from a real value to the corresponding integer.



If we take the random value generated in the range 0 to 1, multiply it by the number of integers we want to generate (in this case, 6), add the lowest integer of the range (in this case, 1), and truncate the result, we will get precisely the mapping of integers onto subranges illustrated in our frequency diagram:

```
***** GENERATE A RANDOM INTEGER FROM 1 TO 6 *****
NROLL = 6*RAN(N) + 1
```

Similarly, if we want to generate a random integer in the range from IA to IB, we apply the same technique: multiply the random real by the number of integers desired ( $IB - IA + 1$ ), add the low value of the range, and truncate the result:

```
***** GENERATE A RANDOM INTEGER FROM IA TO IB *****
NUMB = (IB - IA + 1)*RAN(N) + IA
```

Let us take a look at a simple example of a simulation program using such a random number generator. You want to write a game of "Guess the Number" that you can play against the computer. The computer will randomly pick an integer

in a specified range (you can allow the player to specify the range as input), and the player must then try to guess the number chosen. If the player makes 5 incorrect guesses, then the game is lost. The computer will give the player feedback as to whether the guess is too high or too low, which should be very helpful. Of course, if the player guesses the number correctly, the game is won; the computer then asks if another game is desired. We will use the last formulation we discussed, to generate a random integer in the range from IA to IB.

```
***** GAME OF "GUESS THE NUMBER" *****
CHARACTER ANS
PRINT*, 'HELLO. WELCOME TO GUESS THE NUMBER.'
PRINT*, 'PLEASE ENTER THE DATE - MONTH, DAY, YEAR'
READ*, MONTH, ID, IY
***** INITIALIZE THE SEED TO THE GENERATOR *****
NS = IY*10 + ID*13 + MONTH*3 + 7
5 CONTINUE
    PRINT*, 'ENTER TWO NUMBERS BETWEEN WHICH THE VALUE YOU '
    PRINT*, 'WILL BE GUESSED SHOULD LIE'
    READ*, IA, IB
    N = (IB - IA + 1)*RAN(NS) + IA
    NWRONG = 0
10 CONTINUE
    PRINT*, 'GUESS A NUMBER'
    READ*, NG
    IF (NG. EQ. N) THEN
        PRINT*, 'CONGRATULATIONS! YOU GUESSED IT!'
        GO TO 20
    ELSEIF (NG. GT. N) THEN
        PRINT*, 'THAT IS TOO HIGH'
    ELSE
        PRINT*, 'THAT IS TOO LOW'
    ENDIF
    NWRONG = NWRONG + 1
    IF (NWRONG .LT. 5) GO TO 10
    PRINT*, 'SORRY. YOU LOSE.'
20 PRINT*, 'DO YOU WANT TO PLAY AGAIN? ''Y'' OR ''N'' '
    READ*, ANS
    IF (ANS .EQ. 'Y') GO TO 5
    STOP
END
```

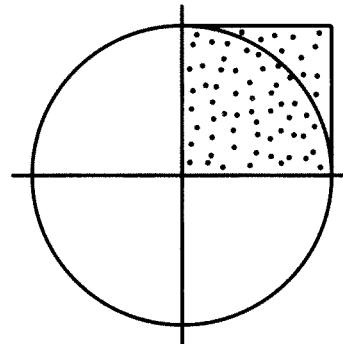
You can see how this method of using random numbers could be adapted to many applications. You can simulate entering members to a queue, or the flip of a coin, or any random event.

## Monte Carlo Techniques

The Monte Carlo method is a technique of using random sampling to solve a deterministic problem that is otherwise too complex to admit of a solution. The technique can be used, for example, for approximating a difficult integral (the area under a curve). To take a simple example dealing with approximating pi, consider a circle of unit radius, and concentrate on the quarter-circle in the upper-right quadrant. Since the area of the entire circle is  $\pi r^2$ , the area of the quarter-circle will be  $\pi/4$  for a circle of radius 1. If we can come up with a value for the area of the circle, we have an approximation for pi.

To do so, we construct a unit square enclosing the quarter-circle, and then "pepper" the square with randomly selected points. Some of these points will fall inside of the circle, and the fraction that do should be proportional to the ratio of the area of the quarter-circle to the area of the unit square. The more points we generate, the better the approximation (of course, in the limit of an infinite number of points, it would be exact). Since we know the area of the unit square, and we can empirically determine the fraction of points that fall inside the circle, we have a way to approximate pi. We have constructed a FORTRAN program which will generate 10,000 randomly selected points in the unit square, count how many of them fall inside the quarter-circle, and thus calculate an approximation to pi.

Notice that the random number from the uniform generator, in the range  $0 \leq x < 1$ , is perfect for a coordinate of a point in the unit square, and does not have to be scaled; we generate two random coordinates X and Y, and then calculate the distance of the point they define from the origin, to see if the point lies in the circle. The distance of the point from the origin is actually the square root of  $X^2 + Y^2$ , and if it is less than 1, the point lies in the circle. But if the square root is less than 1, the value of  $X^2 + Y^2$  will also be less than 1 (and vice versa), which means we can skip the extra computational step of taking the square root, and just compare  $X^2 + Y^2$  to 1.



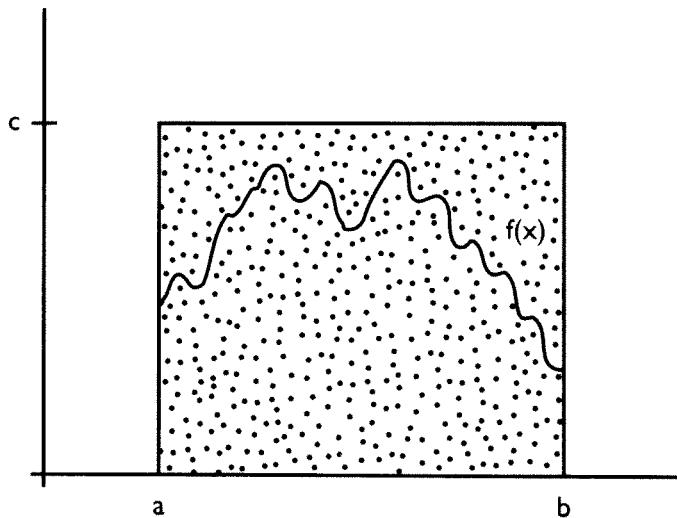
```
*** MONTE CARLO APPROXIMATION OF PI ***
*** SEED THE GENERATOR ***
PRINT *, 'ENTER A 7-DIGIT INTEGER'
READ *, NS
KC = 0
DO 66 I = 1, 10000
  X = RAN(NS)
  Y = RAN(NS)
```

```

        D = X**2 + Y**2
        IF (D.LT.1) KC = KC + 1
66    CONTINUE
PI = 4*KC/10000.0
PRINT *, 'PI IS APPROXIMATELY', PI
STOP
END

```

This technique can be applied to calculating the value of any integral, in the same way. Just create a rectangle (or a cube, etc.) around the area the integral occupies, fill the space with randomly selected points, and count the number of points that fall inside the curve you are integrating. For example, if we have the function  $f(x)$  as some unusual function of  $x$ , and we want to integrate it from  $a$  to  $b$ , we simply pick a height  $c$  for our rectangle which is greater than the maximum of  $f(x)$  from  $a$  to  $b$ , generate random points in the interval, and count how many points fall under the curve. The  $X$  coordinates will be random



values between  $a$  and  $b$  [ $X = (B - A)*RAN(NS) + A$ ], and the  $Y$  coordinates will be random values from 0 to  $c$  [ $Y = C*RAN(NS)$ ]. To see if the point lies under the curve, calculate  $f(X)$ , and see if it is less than the random  $Y$  coordinate generated; if it is,  $Y$  is above the curve. The area of the rectangle is  $C*(B-A)$ , and the value of the integral is the fraction of points generated which lie under the curve times the area of the rectangle.

```

***      MONTE CARLO INTEGRATION OF ARBITRARY F(X)      ****
***      DEFINE F(X) IN A STATEMENT FUNCTION      ***
F(X) = ..... {whatever function is desired}
***      SET THE PARAMETERS A, B, AND C      ***
A = ....
B = ....
C = ....
***      SEED THE GENERATOR      ***
PRINT*, 'ENTER A 7-DIGIT INTEGER'
READ *, NS

```

```

BA = B - A
KC = 0
DO 75 I = 1, 10000
    X = BA*RAN(NS) + A
    Y = C*RAN(NS)
    FUN = F(X)
    IF (Y.LT.FUN) KC = KC + 1
75  CONTINUE
AREA = KC*BA*C/10000
PRINT 88, AREA
88  FORMAT(' THE INTEGRAL FROM A TO B IS ABOUT', F8.3)
STOP
END

```

Note that the function definition and the values of a, b, and c must be put into the above program. You could also modify it to be a subroutine, and make the name of a function of one variable and the values A, B, and C all arguments to the subroutine. The function names actually passed as arguments would then have to be declared as EXTERNAL or INTRINSIC in the calling programs (see Chapter 9, the section on subprograms as arguments).

The accuracy of a Monte Carlo approximation is proportional to the square root of the number of points used; thus to double the accuracy, you have to use four times as many points. There are many other, probably more efficient, methods for approximating simple integrals. But when you get into integrals of higher dimensions, other evaluation techniques may not exist, and a Monte Carlo approach can get you an answer.

Monte Carlo methods are not only used for static situations such as evaluating integrals; they may also be applied to dynamic problems of considerable complexity. In fact, the origin of the term "Monte Carlo" for this type of estimator was the Los Alamos Laboratory. Scientists were faced with the problem of predicting the behavior of neutrons travelling through various kinds of materials. Basic information on the behavior of an individual neutron was available—the laws regarding its interaction with other elementary particles of differing cross-sections, its probability of being deflected or absorbed, its free path length (average distance between collisions), how much energy it would lose in a collision, the probable angle and relative speed at which it would be deflected given an entering speed and angle, and so on. But to handle this theoretically and come up with an equation for the behavior of a huge collection of these neutrons was impossible, and experimental trials were dangerous.

John von Neumann and Stanislaw Ulam came up with the notion of approximating the "life histories" of a group of representative neutrons in a medium by simulating each of their paths by appropriate random selection of time to collision, determination of whether it was absorbed or, if deflected, at

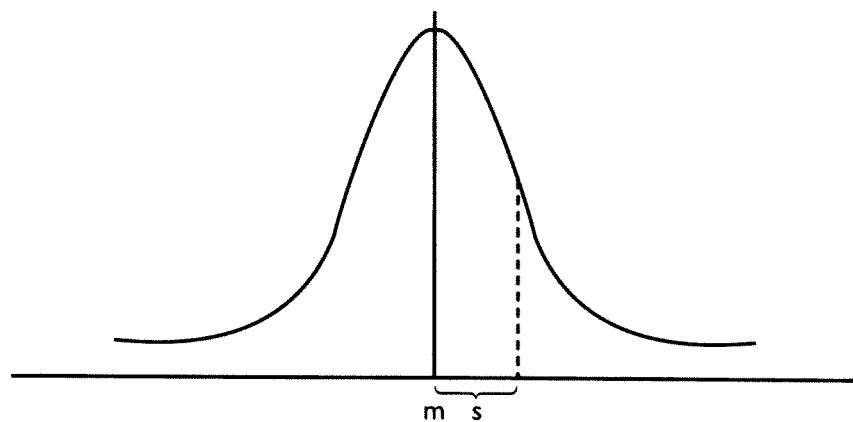
what angle, and other factors. By simulating a large number of such neutron histories, using random selection of values as if they came from a roulette wheel (hence the name "Monte Carlo"), the scientists could gain a pretty good approximation of the percentage of neutrons that would escape from the medium.

Since at the time during and immediately after the Second World War, computers were not very advanced, Ulam says he joked about hiring people, putting them on a boat with abaci, or dice, and having them produce random numbers for the proposed Monte Carlo approximation. He is careful to point out that this method does not give an exact answer, but one good within some tolerable error. Ulam remarks that "Monte Carlo is common sense applied to mathematical foundations of physical laws and processes."

## Other Distributions

So far, all of the illustrations we have examined have been appropriately simulated using a uniform distribution; that is, all of the events were equally likely to occur. However, not all situations in the real world fit into that category. We will examine two other common distributions here—the normal and the exponential—and see how they can be simulated.

The *normal*, or *Gaussian*, distribution is one in which the values cluster fairly closely around the mean (average) value, forming what is called a "bell curve." The distribution is such that 68.26% of the values fall within one standard deviation ( $\pm s$ ) of the mean ( $m$ ). The distribution is represented by the following diagram:



and the equation for the distribution is:

$$f(x) = \frac{1}{s\sqrt{2\pi}} e^{-\frac{1}{2}((x-m)/s)^2}$$

To generate a normally distributed random value, you can use values from a uniform random number generator. If you add up 12 values ( $x_i$ 's) from the random number generator, and then subtract 6 from the sum, you will get a normal random variable  $y$ , from a distribution with a mean of 0 and a standard deviation of 1:

$$y = \left( \sum_{i=1}^{12} x_i \right) - 6$$

If you want to turn this normal random variable ( $y$ ) into one ( $y'$ ) from a distribution with a mean  $m$  and standard deviation  $s$ :

$$y' = sy + m$$

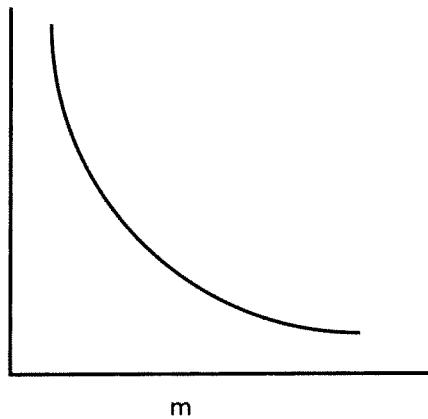
Another method for calculating a pair of Gaussian (normal) variables ( $x_1$ ,  $x_2$ ) from a pair of uniform random variables ( $a$ ,  $b$ ) which have been calculated as described is:

$$\begin{aligned} x_1 &= (-2 \ln a)^{1/2} & \cos(2\pi b) \\ x_2 &= (-2 \ln b)^{1/2} & \sin(2\pi a) \end{aligned}$$

An *exponential distribution* represents events whose probability of occurrence in a short period of time is small, and which are independent of other events (such as equipment failures, wars, and phone calls). The times between such events are said to be exponentially distributed. The equation for such a distribution, with mean  $m$  and variance  $m^2$ , is:

$$f(x) = \frac{e^{-x/m}}{m}$$

and the distribution looks something like the following:



You can easily convert a value  $x$  from a uniform random number generator into a value from an exponential distribution with a mean value  $m$ , by the following equation:

$$y = -m (\ln x) \quad \{x \neq 0\}$$

## Generating Random Test Data

When you write a program that is not self-contained, but rather is designed to accept a range of input data from some external medium and process it, you want to be sure to test the program thoroughly before putting it into production. First you will probably run some simple values through the program so you can hand-check the results. After this is done, you should subject the program to a variety of input data, to be sure that it is "robust," and does not "blow up" or give unreasonable answers for certain input values. One way to do this is to prepare and enter by hand (from the terminal, or enter the data on tape or disk to be read in) reams of data to test your program. However, this is very tedious, time-consuming, and not very creative. Why not let the computer do the work for you?

At the beginning of your program, simply insert a short, removable section that will generate random test data in an appropriate range for the problem, and make it available to the calculational part of the program. Not only is this much easier to do, it is a better test of the robustness of your program (since you might unconsciously be biased in the data you would enter, not wanting your program to blow up) and is also much faster. Data entered from an external source is always much slower than internal arithmetic processing in the program. Thus randomly generated values will feed into the program much faster than values that must be read in from tape or disk (and certainly much faster than typing them in from the terminal!).



## INTERACTIVE PROGRAMS

We have already seen an example of an interactive program in the "Guess a Number" example. Some programs are self-contained, requiring no external input. Others may read in a collection of data and process it, but the data itself really has no impact on the way in which the program executes. In an interactive program, information is input during the course of program execution which can actually change the way in which the program logic progresses. Depending on how the guesses are made, the program will behave in different ways. One can write a whole variety of game-playing programs, from tic-tac-toe to chess and Go. In all of these cases, the program's behavior will depend upon the inputs it receives. Interactive programs are also written to control real-time situations in nuclear power plants, or in missile or satellite guidance, and the like.

In an interactive program, you will be testing the input data and reacting to it. You will also want to be sure to test if the input is legal and meaningful, and not have your program "blow up" due to an illegal input. We did not include that sort of protection in the "Guess a Number" example, but it should

be added. You need the special I/O capabilities discussed in Chapter 11 to handle such situations. Suggestions for recovering from bad user input are given in Chapter 15, in the section on software engineering.

As an interesting exercise at this point, you might want to develop an interactive program so that a user can play "Hangman" against the computer. Develop a dictionary of words for the computer to choose among; you could store these most efficiently in an array, by using a DATA statement. Since the character array will have to be set for the maximum length of any word in the array, some words will be shorter than the maximum length and will be blank-filled. In order to determine the actual length of the word selected from the array, you will have to count its characters until you encounter blanks. The word should be selected *randomly* (now you know how to do this) from the array. Once the word is selected and its actual length is determined, you should display the appropriate number of underscores to indicate the length of the word:

— — — — —

Then the player would begin entering a character guess. If it fits anywhere in the word, you should display *all* of its occurrences, along with any other letters that had been correctly guessed previously, leaving underlines for the remaining letters. We suggest a character variable or array for containing the good guesses. You would then have to test if the word is completed yet. If so, output "Congratulations!"; if not, allow the player to guess again. If the guess is wrong, add 1 to the counter for bad guesses, and see if the player is hanged yet; if not, he or she can guess another letter. Allow play of several games.

If you wish at some point to have printed output from this program but still need output to the screen to play the game, use the technique discussed in Chapter 11 on writing to the printer and the screen in the same program (that is, using duplicate WRITE statements, one to the screen and the other to a file you have opened, which can be sent to the printer later).

Although the simulation described is amusing, the techniques needed to implement it will carry over readily to such scientific cases as interactive flight simulations.



## PATTERN RECOGNITION

The area of *pattern recognition* is interesting and complex, with applications to computer vision ("machine perception") as well as to problems in military intelligence. A robot equipped with a good visual pattern recognition system would be effective in automated industrial work, handling radioactive materials, exploring in areas deadly to humans, and many other ways.

Pattern recognition techniques are very complex and varied, but we will explore a few simple ideas here that can be carried out using FORTRAN

programs. Some simple techniques are using a *template*—that is, a sort of “overlay,” like a stencil, to see if it *fits* the object we are trying to recognize; using centers of gravity and moments about them to identify a pattern’s “signature”; and techniques that will find a best-separating line or plane between identified patterns (“learning”). More complex are brain-like “neural network” devices (“perceptrons”), which “learn” to discriminate patterns by reinforcing certain connections in the network and inhibiting others.

Let us look at the notion of a *template* to recognize a pattern. We need to have a *standard pattern* already in memory, and see how closely it matches a new pattern we are trying to identify. We will assume here that our template and our unknown pattern are both stored as two-dimensional arrays. The values in the arrays are measures of visual intensity of a digitized image at a particular point of the picture (a zero or near-zero value represents background, not a pattern), and the maximum intensity will be 1 (values have been normalized). We will also assume, for the purposes of this exercise, that the orientation of the template and the pattern are standardized, and that they are of the same size.

In trying to match our unknown pattern against a number of standard templates, we need a measure of how much *error* there is in each match. We can compute this as a sum of differences in intensity (array value), squared to get rid of the effect of the sign of the difference, over the area of the two arrays:

$$\text{Error}_{AB} = \sum_i \sum_j (A_{i,j} - B_{i,j})^2$$

That is, the array A contains a template (its two-dimensional pattern of intensity values), and B is the unknown pattern. We compare the pattern to a number of standard templates, and choose the template that gives us the minimum error.

If the template is smaller than the visual field array for the unknown pattern, so the pattern must be matched “somewhere” in the field, we can write code to translate the template across the pattern array, choosing as the best match the position of template on array where the error measure is a minimum. Assuming we have already written the code for calculating the error measure described earlier, if we had an M x M template to move across and down an N x N array representing the picture in which the pattern is to be found, we could do it as follows:

```

PARAMETER (M = 10, N = 40)
REAL A(M,M), B(N,N), ERROR, SMALL
SMALL = 100000.0
LAST = N - M + 1
LENGTH = M - 1
*      TRANSLATES TEMPLATE DOWN PICTURE          *
DO 30  JJ = 1, LAST

```

```

*      TRANSLATES TEMPLATE ACROSS PICTURE      *
DO 20 II = 1, LAST
    DO 20 I = II, II+LENGTH
        ERROR = 0.0
        DO 10 J = JJ, JJ + LENGTH
            ERROR = ERROR + (A(I,J) - B(I,J))**2
10      CONTINUE
        IF (SMALL.GT.ERROR) SMALL = ERROR
20      CONTINUE
30      CONTINUE

```

The smallest error value (SMALL) for this template (that is, the value where it matched the picture best) can then be compared to that of the other available templates, to find the best match.

Assume now that we are trying to recognize handwritten characters, which are capital letters of the alphabet. We will adopt a different technique here, that of determining the center of gravity of the array elements which represent the pattern (these will either be all 1s, as opposed to 0s for background), or they will be values which exceed a certain threshold T. Since the patterns to be matched may be of different sizes (some larger than others, but all displayed on the same-size "screen," which gives us our digitized array of values), we must normalize the pattern measure by dividing through by the area of the rectangle it occupies. We will determine this rectangle by finding the four *extremum* points of the pattern—that is, the smallest and largest x(i) and y(j) coordinates which define points of the picture which exceed the threshold T.

To find the points which define the rectangle:

```

PARAMETER (N = 20)
REAL A(N,N), T
DATA IMIN, JMIN / 2*100000 /, IMAX, JMAX / 2*0 /
DO 40 I = 1, N
    DO 40 J = 1, N
        IF (A(I,J).GE.T) THEN
            IF (I.LT.IMIN) IMIN = I
            IF (I.GT.IMAX) IMAX = I
            IF (J.LT.JMIN) JMIN = J
            IF (J.GT.JMAX) JMAX = J
        ENDIF
40    CONTINUE
RECT = (IMAX - IMIN) * (JMAX - JMIN)

```

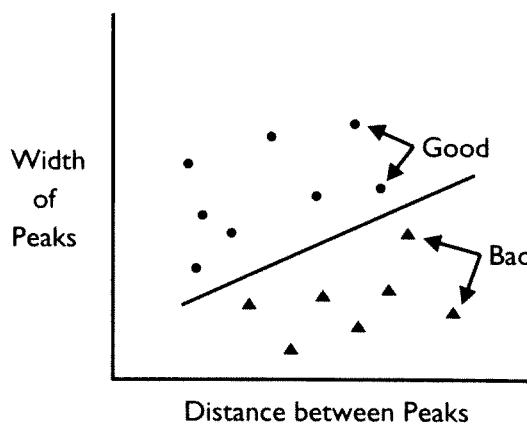
To find the *center of gravity* of the pattern, add up the sum of all the x-coordinates (I-values) of points in the array which exceed the threshold T, and the sum of all the y-coordinates (J-values) of these points, and divide by the

number of points that exceeded the threshold. You can write code to do this as an exercise. Call them XBAR and YBAR.

Once the center of gravity and the size (RECT) of the pattern have been found, its “signature” can be determined. This will be the sum of all the “moments” of the points in the pattern which exceed the threshold about the center of gravity, divided by the area of the pattern (RECT), to normalize in order to eliminate the different-size factor. The “moment” of a point about the center of gravity (c.g.) is its distance from the c.g. times the value of the array at that point (the value must exceed T). Write code to calculate the sum of the moments.

The sum of the “moments,” divided by the area of the pattern (RECT), should give you a *signature* for the pattern. The signature for a printed “A” should be significantly different from that of a printed “C”, and so on. These signature values should help you recognize and differentiate the characters. Notice that this technique can be applied equally well to patterns other than letters of the alphabet.

Another method of pattern recognition is that of “learning” to distinguish between two types of identified patterns by taking measurements on them and then finding the best-fitting line (or plane of 2 or more dimensions) that separates them. Imagine that you have data on audio signals for a number of good jet engines, and data for several faulty jet engines. Assume that you take two representative measures (say, distance between peaks in the signal, and average width [one standard deviation] of peaks) for each jet engine. When they are plotted on a two-dimensional graph, you find that the good jet engine points on the graph cluster in one place, and the bad jet engines cluster together in a different area of the graph (this is a best-case scenario).



You can then find the best-fitting line that separates these two sets of points. This can be done with a variation on the least-squares method of fitting a line to data, a concept that will be discussed in the next chapter. Once the best-fitting line has been found, of equation  $y = ax + b$  (your program has determined a and b), this can then be used to classify any new unknown jet

engine signal as *good* (falls above the line), or *faulty* (falls below the line). Notice that this technique can be extended to handle more than two measures on the data. If three measures are taken, you will find the best-fitting (two-dimensional) plane that separates the data points, and if you take  $n$  measures on the data, you will find the best-fitting  $n-1$ -dimensional surface that separates the points. Probably the more meaningful measures you can make of the data, the better the recognition will be.

*Neural networks* (or perceptrons) are a bit too complex to get into here, but the idea is to build a computer model of a “brain” that can then “learn” different patterns by reinforcing connections in the network which lead to a correct classification and inhibiting those that lead to incorrect classifications. This kind of network can also be used to “group” certain patterns together (that is, recognize similarities). This has currently reemerged as a “hot” topic; for more information on this kind of modelling, see the references at the end of the chapter.



## IMPLEMENTING OTHER DATA STRUCTURES

A wise man (Niklaus Wirth) once said that it is a combination of algorithms and data structures that makes up programs. We have spent considerable time in this book discussing and implementing various algorithms, but we have not given comparable time to data structures. We have discussed what is available in our language, FORTRAN: a number of different data types (integer, real, double precision, complex, logical, and character), and basic array structures that may have from one to seven dimensions. These have seemed pretty adaptable to the problems we have discussed, but might it be that we have chosen the problems carefully so the structures *will* work?

### Abstract Data Structures

A much-discussed topic in the computer science area these days is that of *data abstraction*. By this is meant, broadly, the ability to think about the type of data you are going to deal with and the operations you want to perform on it, without having to worry about how it will actually be implemented (carried out) in a particular language or on a particular machine. Such a data structure, conceived in this implementation-independent manner, is referred to as an *abstract data type* (ADT). An ADT is defined in terms of the *kind of objects* that are in the type (its domain) and the range of *operations* that will be performed on it. Only after all of this has been clearly thought out, the ADT defined and structured, and its operations identified, should any thought be given to how it will be actually implemented.

If an ADT is carefully developed, in a systematic manner, it will be *reusable* just as good structured program procedures are reusable. The purpose of this

approach is to ensure that the structure you have designed is the one most appropriate for the problem at hand, rather than having your thinking about the problem constrained by worrying about the limitations of what the programming language can do. If the properties of *inheritance* and *identity* are added to an ADT, we have the basis for *object-oriented* programming (see references at the end of the chapter).

For the FORTRAN programmer, the two-dimensional array can serve as a simple introduction to the idea of an ADT. For most of the time that we use a two-dimensional array in FORTRAN, we think of it in terms of a table or a matrix, organized in a 2-dimensional fashion; that is, we are not concerned about how it is actually implemented on the machine. This is actually the most effective way for us to use the 2-D array. At some point—perhaps when we are thinking about efficient I/O for the structure—we may look into its implementation, and be surprised to discover that it is organized as a linear list with the 2-D structure mapped column-by-column onto this list.

We will look into several variations on this approach to problem solving using ADTs, and then how we can best approximate implementing them for actual use in FORTRAN.

## User-Defined Types

Several newer languages, such as Pascal and Modula-2, have the feature that the user (programmer) can actually *create* some new data types to suit certain problems. The actual construction of these types gives them a somewhat limited usability, since they cannot be used in I/O, and the comparisons involving them are very limited. However, they do allow the programmer to think in terms of categories that are not explicitly part of the programming language base. For example, we have done several problems in terms of months of the year, or days of the week. It would have been useful if these explicitly existed in FORTRAN as data types we could work with directly. Because they are ordered (the first, the second, etc.), we could talk about their relative positions in the ordering, or compare them for equality.

Pascal and other languages have allowed the programmer to define a new “type” for such occasions. It is usually of a form something like:

```
type
  days = (Sunday, Monday, . . . );
```

and these values can then be assigned to variables which have been declared to be of the type days. They can be compared for equality or according to order and, since they are ordered, they can be used as subscripts for arrays. But they cannot appear in I/O statements, and their general utility is limited.

These *enumerated* types are constructed to allow the programmer to have the illusion of manipulating data objects on the same level as integers and

reals, which are actually just names of objects grouped under a class name. We can create an array, with an identifiable name, which contains the names of these data objects, and use it in almost the same way. Since their names are character strings, we make our array of type character, and store the object names in it.

```
CHARACTER*10 MONTH(12), DAYS(7)
DATA MONTH / 'JANUARY', 'FEBRUARY', 'MARCH', . . .    /
DATA DAYS / 'SUNDAY', 'MONDAY', 'TUESDAY', . . .    /
```

Granted, we cannot run a loop from Monday through Friday, but we know enough about our data objects and their order to run a loop:

```
DO 10 I = 2, 6
    WEEK = DAYS(I)
    . . .
```

and then do whatever operation was desired with the variable WEEK. We can assign the string 'SUNDAY' (or DAYS(1)) to any character string variable. We cannot use March as a subscript, but we can use its position in the MONTHS array (3) as a subscript for another array. And we have the added advantage not available in these other languages of being able to use these data objects in I/O statements.

Thus, we have created a collection of data objects known as *months* (a new "type"), and have stored them in the array called MONTHS. We can determine the relative ordering of two months by consulting the array. We could construct a logical function which would return .TRUE. if two month names passed to it were in proper order (the first less than or equal to the second), and return .FALSE. otherwise. In fact, we could construct a general-purpose logical function that would accept *any* array containing a "user-defined type," its length, and two objects from the type, and return the correct logical value representing their proper ordering. We are thus beginning to increase our set of ADT tools (operations) as well as creating new data types.

```
LOGICAL FUNCTION ORDER (ADTYPE, N, ELONE, ELTWO)
** THIS FUNCTION WILL DETERMINE WHETHER TWO ELEMENTS FROM AN **
** ABSTRACT DATA TYPE ARE IN PROPER ORDER OR NOT   **
CHARACTER*(*) ADTYPE(N), ELONE, ELTWO
INTEGER NONE, NTWO
NONE = 0
NTWO = 0
I = 1
20 CONTINUE
```

```

        IF (NONE.EQ.0 .OR. NTWO.EQ.0) THEN
            IF (ADTYPE(I).EQ.ELONE) THEN
                NONE = I
            ELSEIF (ADTYPE(I).EQ.ELTWO) THEN
                NTWO = I
            ENDIF
        ENDIF
        I = I + 1
        IF (I .LE. N) GO TO 20
        IF (NONE.EQ.0 .OR. NTWO.EQ.0) THEN
            PRINT*, 'THE ELEMENTS ARE NOT BOTH IN THE TYPE!'
            ORDER = .FALSE.
            RETURN
        ENDIF
        ORDER = NONE .LE. NTWO
        RETURN
    END

```

Similarly, to take care of the case where we would like to think of the month names as identifying rows of a two-dimensional array (perhaps a table of daily expenses, organized month by month), we will use *parallel arrays* to implement the structure. Our EXPENS array has 12 rows and 31 columns; each row represents a month of the year, and we carry this connection through by considering our MONTH array to be in parallel with the EXPENS array; the Nth row of the EXPENS array corresponds to the Nth month in the MONTH array.

Thus, if in a loop we want to look at the average expenses for the Ith month, we will use parallel arrays MONTH and NDAYS to print out the month name and determine how many days it has:

```

REAL EXPENS(12, 31)
CHARACTER*12 MONTH(12)
INTEGER NDAYS(12)
DATA NDAYS / 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31/
...
DO 50  I = 1, 12
    N = NDAYS(I)
    SUM = 0.0
    DO 40  J = 1, N
40        SUM = SUM + EXPENS(I,J)
        PRINT*, 'THE AVERAGE EXPENSES FOR THE MONTH OF ', MONTH(I)
        PRINT*, 'WERE $', SUM/N
50    CONTINUE

```

Similar feats can be accomplished by considering arrays of character names as representing "new" data objects to be manipulated in your programs, using parallel arrays and the like.

## Fortran 90 Derived Data Types

Fortran 90 will give the programmer the capability to define (construct) new data types out of the existing six available types in the language. A type definition has the form:

```
TYPE typename
...
END TYPE typename
```

This can be used to combine several related values under one reference. For example, if you are doing work on chemical compounds, in which you have to handle the name of the compound, an integer number representing the number of days it has been in the laboratory, and a real value representing the amount of the substance (in grams), you could construct a COMPOUND type:

```
TYPE COMPOUND
    CHARACTER (30) NAME
    INTEGER DAYS
    REAL AMOUNT
END TYPE COMPOUND
```

Then variables could be made to be of the type COMPOUND:

```
TYPE (COMPOUND) :: ALPHA, BETA, GAMMA
```

Reference can be made to a part of the substructure of a derived data type by using the % symbol; for example,

```
ALPHA%NAME = 'H2O'
PRINT*, GAMMA%AMOUNT
```

Operators can also be defined that can be used with derived data types. For example, a function COMBINE could be defined (an exercise left to the reader) which would give the resulting compound structure when two compounds are combined:

```
FUNCTION COMBINE (A, B) RESULT (NEW)
    TYPE (COMPOUND) :: A, B, NEW
    ...
END FUNCTION COMBINE
```

and this function could then be given as an alternate meaning to the operator '+' when used with respect to two compounds:

```
INTERFACE OPERATOR (+)
    FUNCTION COMBINE (A, B) RESULT (NEW)
        TYPE (COMPOUND) :: A, B, NEW, COMBINE
    END FUNCTION COMBINE
END INTERFACE
```

which would then give meaning to an expression such as

ALPHA + BETA

as an alternate expression for

COMBINE (ALPHA, BETA)

This is referred to as *overloading* the operator (in this case, +).

## Records

Occasionally, you want to group data together that is not of the same type. For example, you might want to write out to disk information on substances you are testing, which include the name of the substance, its molecular weight, the number of tests you made on it, and the date. These are, respectively, character, real, integer, and character pieces of data, yet you want to consider them as grouped together. Each substance name, weight, or other descriptor is an element in an array that has a total length equal to the number of substances you are testing. You write the information all out in one record, but it would be nice to have a way of considering it all to be grouped together.

All you can do in FORTRAN 77 is to have these arrays "parallel" to one another, and all of the same length, and then write an element from each out in the output statement. You can indicate by *comments* the group relation among them.

```

CHARACTER SUBS(50)*15, DATE(50)*8
REAL WEIGHT(50)
INTEGER NTESTS(50)

.
.

***** SUBSTANCE TESTING RECORD CONTAINS..... *****
***** SUBS - SUBSTANCE NAME *****
***** WEIGHT - MOLECULAR WEIGHT *****
***** NTESTS - # OF TESTS MADE ON SUBSTANCE *****
***** DATE - DATE OF THE TESTS *****

DO 30 I = 1, 50
      WRITE (3, 33) SUBS(I), WEIGHT(I), NTESTS(I), DATE(I)
etc.

```

A language such as Pascal would allow you to have a single record name to identify the group of related data, which would simplify references to the entire structure. However, we can make do in FORTRAN 77 by using comments and keeping track. Fortran 90's defined types will allow the programmer to create and name record-like structures, as is obvious from our discussion in the previous section. For our example here, we would simply

construct the type RECORD:

```
TYPE RECORD
  CHARACTER*15 SUBS
  REAL WEIGHT
  INTEGER NTESTS
  CHARACTER*8 DATE
END TYPE RECORD
```

and we then could declare an array EMPLOY of length 50 to be of type RECORD:

```
TYPE (RECORD) :: EMPLOY(50)
```

## Sets

Occasionally a problem is most meaningfully formulated in terms of *sets*, collections of items in which there are no duplicates. The *union* of two sets will then tell you all of the elements that are in either one or both, and the *intersection* of two sets will tell you what they have in common. A limited set-handling capability is available in Pascal, but it restricts the size that sets may have, and does not allow I/O on them. We can extend our simulation of user-defined types to include means of representing and manipulating sets in FORTRAN. Sets could be treated simply as represented by arrays, and we could then work with the arrays to find unions and intersections. For example, if we have two integer arrays SETONE and SETTWO, we could just write code, by searching through the arrays, to fill a third set (SET3) with the elements in the union of the two, and a fourth set (SET4) for the intersection. Think about how you would write this as an exercise.

There is another way to view sets, which allows a more general representation. The sets we are dealing with might be, say, collections of disease symptoms. There is a "universal," *base* set of all possible identifiable symptoms, and so we set this up as a character array containing all by name. We also set up a parallel logical array with as many entries as the symptom array for each "set" we are looking at. A particular set, FLU, will have all of the symptoms associated with flu, but none of the others. Another set, GOUT, will contain the symptoms for gout; and so on. Instead of just having a short array containing the list of symptoms under each set, such as FLU, it is represented by a logical array paralleling the SYMPTM array, with the value .TRUE. stored in each location where a symptom on the list fits the disease, and .FALSE. elsewhere.

```
CHARACTER*12 SYMPTM(30)
LOGICAL FLU(30), GOUT(30), HEART(30), UNKNWN(30), INTER(30)
DATA SYMPTM/ 'CHILLS', 'FEVER', 'HEADACHE', ... /
```

In this way, if we want to know what characteristics two diseases (such as FLU and MLARIA, for example), have in common, we would just look at the *intersection* of the two, by *anding* all of the elements in their arrays together:

```
DO 50 I = 1, 30
50    INTER(I) = FLU(I)*MLARIA(I)
```

Then if we wanted to know what symptoms these were, we would print out all of the entries in SYMPTM for which there was a .T. in the parallel array INTER.

Similarly, an unknown disease could be compared with all of the known diseases, and be judged to be most like the one for which the overlap was greatest (that is, the intersection had the highest number of .TRUE.s). As a matter of fact, if the number in the base set is small enough (as our 30 elements here), the logical values could all be packed into *one* word in memory (as 1s for .TRUE.s, ignored otherwise, times increasing powers of 2). We discussed a similar example in the section on *packing* in Chapter 10. These single variables could then be ANDed or ORed together in one step to find intersections and unions. The words would have to be *unpacked* to compare with the SYMPTM array, as we did in the previous example.

## Stacks

An abstract data type (ADT) that shows up in a variety of problem situations is the *stack*. A stack is a linear list which has additions and removals from the same end on the list; it is a "Last-In-First-Out" (LIFO) structure, like a stack of trays in a cafeteria. Stacks can be used to keep track of the most recent data, especially if backtracking might be needed, such as in searching a maze, or hiring and firing recent employees. Recursive operations (which we discussed briefly) are implemented using stacks. If a stack structure is appropriate to your problem, what operations would you want to be able to make on the structure? You would want to be able to *remove* ("pop") a value from the top of the list, or *add* ("push") a value onto the top of the list. You might also want to be able to strip off all the entries on the stack, from most recent ("top") to bottom. All your procedures would need would be the list, the pointer to the current top of the list, and which operation was to be performed. For example, a "pop" procedure that removes an item would be:

```
SUBROUTINE POP (LIST, NPTR)
INTEGER LIST(*)
PRINT*, LIST(NPTR), ' IS REMOVED'
NPTR = NPTR - 1
RETURN
END
```

This could be modified to send the value “popped” to a location, or to handle stacks of some other type of data. Note that only now have we looked (and not too closely) at an actual implementation of the stack. Since it is a linear list, it is natural to map it onto a one-dimensional array (our basic building block structure). The length of the entire list space available must be known, so it can be dimensioned, but this is not a concern of the “pop” routine; all it needs is the beginning location of the stack and the pointer to the current top of the stack (which it then changes after the pop).

## Queues

A *queue* is a linear list which has additions to one end of the list and removals from the opposite end; it is a “First-In-First-Out” (FIFO) structure. A queue can be used to represent cars queueing up at a traffic light, or shoppers at the supermarket, or programmers waiting their turn for the CPU in a time-sharing environment. For example, imagine that your problem is to keep track of planes about to take off from a large airport (you are working on automating the job of the air traffic controller). Data comes in that identifies new planes queueing up to take off, which have to be added onto one end of the list; when the opportunity arises, you allow a plane from the opposite end of the list to take off. You thus need a list with *two* pointers, one to each end of the queue. The operations you might want to perform are: *add* a new plane to one end of the list, *remove* a plane from the opposite end, *list* the entire contents of the queue, or simply identify *how many* planes are in the queue at the present moment.

Now that we have clarified what our data structure is like, and what operations we would like to perform on it, we can concern ourselves with implementing it in FORTRAN. Since it is a list, it clearly should be mapped onto a one-dimensional array. We have to be sure to make the array long enough, as in the case of the stack, but it will tend to grow primarily “down,” that is, new entries will be added on behind old ones, so there must be room in the list for them. Thus, initially, it would be advisable to place our *first* entrant into the queue at the *end* of the array.

We need to establish two pointers, one to the **TOP** of the list, where removals will come from, and another to the **BOTTOM** of the list, where newcomers will be added. Initially, in an empty list, the two pointers will both be the same—the length of the list. Here is a program skeleton that establishes a queue (we have made the entries integer, like plane ID numbers) and the pointers, and begins a loop which would continue to read inputs (which indicate ‘ADD’, ‘REMOVE’, or ‘SHUT DOWN’) until the queue is empty or the field shut down.

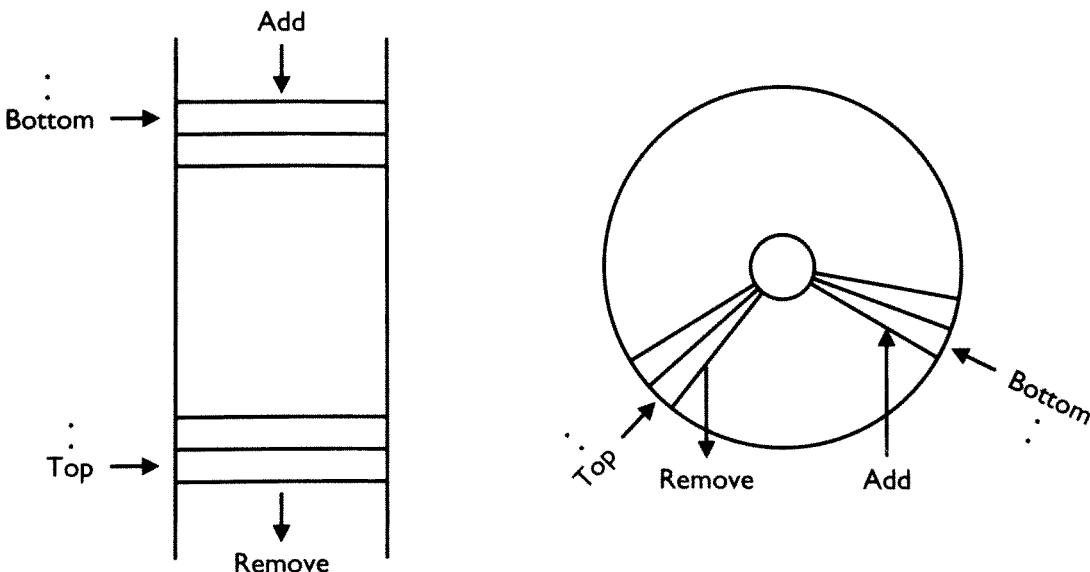
```
PARAMETER (LEN = 100)
INTEGER QUEUE (LEN), TOP, BOTTOM, LEN
```

```

CHARACTER STATUS*9
DATA NUM / 1 /
TOP = LEN
BOTTOM = LEN
READ*, STATUS, ID
40 IF (STATUS.NE.'SHUT DOWN' .AND. NUM.NE.0) THEN
    IF(STATUS.EQ.'ADD')THEN
        QUEUE(BOTTOM) = ID
        BOTTOM = BOTTOM - 1
    ELSE
        PRINT*, QUEUE(TOP), ' CAN TAKE OFF'
        TOP = TOP - 1
    ENDIF
    NUM = TOP - BOTTOM
    READ*, STATUS, ID
    GO TO 40
ENDIF
STOP
END

```

One thing we did not concern ourselves with here was what would happen if the BOTTOM of the list were to drop below the beginning point of the array. One way to handle this is to try to make the array so long that this should never happen. But another way is to make the queue "wrap around" on itself. Since presumably planes are leaving from the top of the list almost as fast as new planes are being added, the top will be moving down the list as bottom moves down. This leaves space available at the physical end of the array. Thus, if BOTTOM were to go to zero, we could move it to the end of the array (so we should test for this after any "add"):



```

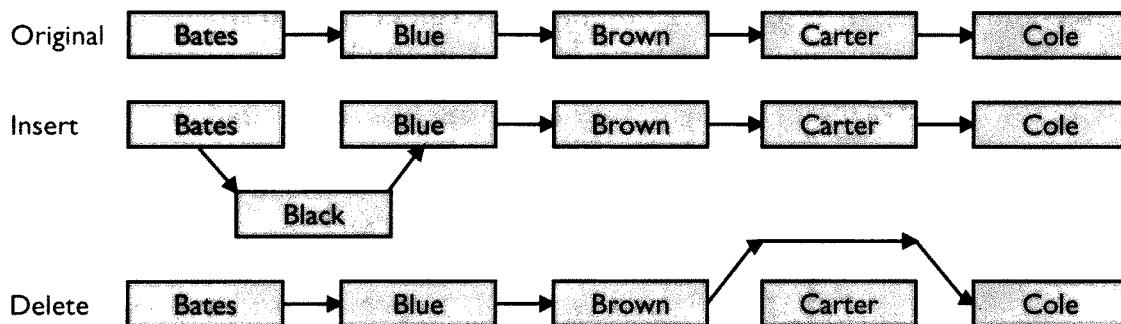
BOTTOM = BOTTOM - 1
IF (BOTTOM.EQ.0) BOTTOM = LEN
IF (BOTTOM.EQ.TOP) THEN
    PRINT*, 'WHOOPS! QUEUE HAS RUN OUT OF SPACE!'
    STOP
ENDIF

```

## Linked Lists

All of our lists (arrays) so far have been *sequentially* ordered. If we wanted to actually *add* or *remove* a value from the middle of the array, all the values in the array with higher subscripts would have to be moved down or up one place, which seems rather inefficient. To maintain a "dynamic" quality for a list, such that it is easy to change, we can establish a list of "pointers" that indicate the *links* from one value in the array to the next. Then if a change (addition or deletion) is to be made, it simply means altering one or two links.

For example, if our list were *linked* to be an alphabetical list of names, we would insert or delete as indicated:



In a sense, then, every entry in the data structure we want to construct has its normal elements *plus one more*—a pointer to the next data element in the collection. There must also be a single pointer variable that points to where the list begins. The pointer associated with the position that is at the end of the linked list will have no pointer; that is, its value will be "null," or zero. The operations we want to perform on a linked list are to insert, delete, or output the list according to the order the links indicate.

In FORTRAN, to construct a linked list, we must have an array of integer pointers which is *parallel* to the structure it links. For example, it would be a great advantage in sorting not to have to move elements of a structure every time an exchange is made, as in the bubble sort. This is especially true if a two-dimensional array is being sorted based on the values in one column, or a

group of parallel, related arrays is being sorted based on the values in one of the arrays. In a straight-forward bubble sort, every time a pair of values in the “key” list or column was interchanged, a pair of *rows* in the 2-D array, or a pair of entries in *each* of the related parallel arrays, would have to be interchanged.

To avoid all of this moving back and forth, if a linked list is used, then only the pointers have to be exchanged each time a pair of values is seen to be out of order, and the pointers are used to determine on the next pass whether adjacent pairs are in order or need to be switched, until no changes need to be made. To appreciate implementing this problem using a linked list, let us look at a short simple example.

Imagine the array of 5 integer values shown, which you want to sort into ascending order. It is part of a group of parallel arrays which are all to be reordered based on arranging this list into ascending order. We begin with a list of pointers, where each pointer for location *n* initially points to the next location it is linked to in the list. Thus the initial values are as shown, with the pointer for entry 1 in the array pointing to position 2, the pointer for location 2 pointing to position 3, and so on. The location (LP) pointing to the “head” of the list initially points to position 1. We have incorporated this into the LINK array as location 0 to make the coding easier. The values shown indicate how the link list values are changed as the bubble sort proceeds. Note that at the end, the link list contains the positions in ARRAY of the values arranged in ascending order (4, 1, 3, 5, 2). Note that if we can get this to work for a list 5 elements long, we can get it to work for a list of any length!

Array	Link																																								
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">3</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">7</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">4</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">2</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">5</td></tr> </table>		3		7		4		2		5	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">1</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">2</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">3</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">4</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">5</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">0</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> </table>		1					2					3					4					5					0			
	3																																								
	7																																								
	4																																								
	2																																								
	5																																								
	1																																								
	2																																								
	3																																								
	4																																								
	5																																								
	0																																								
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">3</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">7</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">4</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">2</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">5</td></tr> </table>		3		7		4		2		5	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">1</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">2</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">3</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">4</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">5</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">0</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> </table>		1					2					3					4					5					0			
	3																																								
	7																																								
	4																																								
	2																																								
	5																																								
	1																																								
	2																																								
	3																																								
	4																																								
	5																																								
	0																																								
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">7</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">4</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">3</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">2</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">5</td></tr> </table>		7		4		3		2		5	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">1</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">2</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">3</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">4</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">5</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">0</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> </table>		1					2					3					4					5					0			
	7																																								
	4																																								
	3																																								
	2																																								
	5																																								
	1																																								
	2																																								
	3																																								
	4																																								
	5																																								
	0																																								
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">4</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">1</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">3</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">5</td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">2</td></tr> </table>		4		1		3		5		2	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">1</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">2</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">3</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">4</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">5</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> <tr><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px; text-align: center;">0</td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td><td style="width: 20px; height: 20px;"></td></tr> </table>		1					2					3					4					5					0			
	4																																								
	1																																								
	3																																								
	5																																								
	2																																								
	1																																								
	2																																								
	3																																								
	4																																								
	5																																								
	0																																								

To implement the sort on ARRAY, we compare “adjacent” array positions as indicated by the LINK array; when an interchange is indicated, we interchange the LINK values, without changing ARRAY (since LINK is keeping track of the order in which the ARRAY values should be arranged).

```

PARAMETER (N = 5)
INTEGER LINK(0:N), ARRAY(N), TEMP
LOGICAL SWITCH

. . .
DO 10 I = 0, N-1
10     LINK(I) = I+1
LINK(N) = 0
20 CONTINUE
    SWITCH = .FALSE.
    DO 30 I = 0, N-2
        IF ( ARRAY(LINK(I)) .LE. ARRAY(LINK(I+1)) ) THEN
            TEMP = LINK(I)
            LINK(I) = LINK(I+1)
            LINK(I+1) = TEMP
            SWITCH = .TRUE.
        ENDIF
30 CONTINUE
    IF (SWITCH) GO TO 20
***** OUTPUT VALUES FROM ARRAY AND RELATED PARALLEL ARRAYS
***** IN ORDER INDICATED BY LINK *****
    DO 40 I = 0, N-1
40     WRITE (*,*) ARRAY (LINK(I))    etc.

```

Your familiarity with and use of implemented ADTs such as stacks, queues, and linked lists will not only make it easier for you to solve problems that are representable by such structures, but will make it easier for you to design and implement abstract data structures of different types when you need them. You are probably beginning to discover that the more programming design tools you design and develop, the better you get at it!



## FORTRAN 90 FEATURES

In this chapter, we discussed the introduction of a random number generator subroutine, RANDOM\_NUMBER, and a subroutine to “seed” the sequence, RANDOM\_SEED, as part of the new Standard. It would be simple to use these to construct a function, if desired, to match non-standard function references in earlier programs.

New data types can be constructed in Fortran 90 out of the existing six data types in the language. These are derived using a TYPE . . . END TYPE structure. Variables can then be declared to be of the new data type:

TYPE (newtype) :: list.

Operations can be defined for use on derived data types by using the INTERFACE and OPERATOR keywords. The mechanisms for defining derived data types and operators were discussed in the chapter, and are also covered in Appendix E.



## SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

We have already looked at *deterministic* models of problems in which change occurs. Many other interesting models involve *probabilistic* simulations of change, and we can program these using *random numbers*. Our random numbers come most easily from random number generators; the mid-square method and the linear congruential method of generating random values were discussed. There usually is a random number generator available on most FORTRAN compilers, though it is not part of the FORTRAN 77 Standard, and the names and methods of reference may differ. Most random number generators are written as functions (though the one which will be standardly provided in Fortran 90 is a subroutine), and most allow a *seed* value to be specified (which changes on successive references to the generator)—for example, a reference to RAN (NSEED).

All of the generators provide a real value (x) that lies uniformly distributed between 0 and 1 ( $0 \leq x < 1$ ); that is, the distribution of many of these values is flat. These values can then be *scaled* to produce numbers to suit the range of values you are simulating:

real value between 0 and A:	$A * \text{RAN}(\text{NSEED})$
real value between A and B:	$(B - A) * \text{RAN}(\text{NSEED}) + A$
integer value between 1 and N:	$\text{INT}(N * \text{RAN}(\text{NSEED}) + 1)$
integer value between M and N:	$\text{INT}((N - M + 1) * \text{RAN}(\text{NSEED}) + M)$

Various examples using such random values were discussed.

A *Monte Carlo* method is one in which random values are generated to simulate, for example, points covering an area that cannot be integrated readily otherwise, or values for many moving particles, as for instance in a nuclear reactor.

Other distributions of random values, such as Gaussian, Poisson, or exponential, can be created using the uniform values. You can use the random generators to create random test data for checking out a program that processes large quantities of information. Random values are useful in simulating games which may be programmed to work interactively (such as "Hangman" or Tic-Tac-Toe).

*Pattern recognition* can be performed by testing how closely a pattern matches a *template*. Also, calculating moments about the center of gravity of a pattern can determine a unique “signature” for that pattern.

An *abstract data structure* (ADT) is one created to suit a problem, without concern as to how it can be implemented on a particular computer. An ADT is defined in terms of the kind of objects that are in it and the operations that will be performed on them. A two-dimensional array, or a user-defined type, are examples of ADTs.

A *record* is a collection of related data, which may be of different types. FORTRAN 77 has no specific record structure, but one can be simulated using the methods discussed in the chapter. A *set* is a collection of related information (of the same type) in which there are no duplicates. We can then look at the *union* or the *intersection* of two sets. A more general representation of sets can be implemented by creating a “universal” reference set, and then logical arrays for the particular sets, indicating whether they contain the members of the universal set (this data could also be packed); then the union of two sets can be determined using the .OR. operator, and the intersection of two sets can be found by using .AND. .

A *stack* is a linear list which has additions and removals from the same end of the list (LIFO); this ADT can be implemented using a one-dimensional array with a pointer to the “top” of the list, onto which values can be “pushed,” or from which they can be “popped.” A *queue* is a linear list which has additions to one end and removals from the other end (FIFO); this can be implemented using a 1-D array with two pointers.

A *linked list* is an array of *pointers* to positions in another data structure. Using such a structure, the structure pointed to can have additions, deletions, and sorts performed without the usual amount of moving data around. An addition or a deletion simply involves making a change in the pointer array. A complex structure can be sorted simply by reordering the pointers in the link array, without moving the complex structure, and then writing out (or storing) the reordered structure according to the new set of pointers in the link array.



## EXERCISES

1. Write a *deterministic* simulation program that will calculate how many years it will take a population to double in size, at various rates of annual increase, from 0.5% to 5%, in steps of 0.1%; create a table of your results, indicating percent increase and doubling time. (The 1985 *World Population Data Sheet* reported the doubling time as 41 years for an annual rate of 1.7%; how does this compare with your results?)

- 2. Write a function which will generate uniform random numbers (normalized) using von Neumann's mid-square method; use integers of 4 digits to square.
- 3. Write a function to generate uniform random numbers using the linear congruential method. Pick 4-digit multiplier and additive factors, and take your result mod 10,000; maintain the integer seed, but return a fractional random number between 0 and 1. Compare the results of your generator to those of the system generator available to you. A good visual way to do this is to have each generate a large number of random numbers (at least 1000) and plot comparative histograms of how many values fall in different uniform subranges of the range (0–1).
- 4. The game of craps, "according to Hoyle," proceeds as follows. At the beginning of each game, a player rolls a pair of nonloaded dice, each of which is equally likely to come up any value from 1 through 6. Write a subroutine or function that will roll the dice for you and return the sum of the rolls (be sure to give your random number generator a seed, which is established in the main program, not the subroutine). On the first roll of the game, if you roll a 2, 3, or 12, you lose and the game is over; if you roll a 7 or an 11, you win. If you roll any of the other six possibilities, the game continues. You continue rolling, trying to "match" the first roll. If you roll a 7, you lose; if you roll a match, you win; otherwise, roll again. Write a program that will play 100 games of craps, and count how many you win and how many you lose.
- 5. How much is a billion dollars? Determine how many years you could go on spending \$10,000 a day before you had spent all of your billion dollars. (This can be done with a straightforward calculation.) Now calculate how long it would take, on a scheme that guaranteed to double your money every year, for you to accumulate a billion dollars, if you begin with one dollar. (If that takes too long, how about settling for a cool million?)
- 6. Porthos and Aramis are going to fight a duel with pistols. Since Porthos is the poorer shot, and only hits what he aims at  $1/4$  of the time, Aramis always lets him shoot first. Aramis hits what he aims at once every 2.5 times. In the duel, they alternate shooting until someone is hit. Simulate 100 of these duels, using a random number generator, and report how many times Aramis is shot and how many times Porthos is shot.
- 7. Write a subroutine to simulate "shuffling" and then "dealing" cards from a deck of 52 cards. One way to "shuffle" is to randomly interchange pairs of cards in the deck for about 100 interchanges, and then deal from the top of the deck. Another way is to randomly select a card from the deck, and then remove it, shortening the deck and noting that you have a deck one card shorter to randomly choose from the next time. One way to do this is simply interchange the card you "dealt" with the card at the current "bottom" of the deck, and then shorten the length of the deck (active part of the array) by one for the next deal.

Begin with a deck that initially has all of the cards in order. You might find it simplest to represent them as the integers 1–52, and worry later about how to convert these numbers into suit and rank designations (the first 13 cards, 1–13, are Spades, in rank order, 2, 3 . . . K, A; then the Hearts; and so on). Print out the contents, in terms of suits and ranks, of each hand dealt.

**8.** Using your deck-dealing routine of the previous exercise, deal out  $N$  hands of 5-card stud poker ( $3 \leq N \leq 10$ ). Now evaluate each of the hands to determine whether it contains a bust, one pair, two pair, three of a kind, a straight (a “run” of cards in rank sequence), a flush (all in the same suit), four of a kind, or a straight flush. This takes some careful coding.

**9.** Deal 1000 five-card stud poker hands, and count how many fall into each of the possible categories mentioned in the previous exercise: bust, one pair, etc. Print out the statistics.

**10.** In ancient Rome, many gladiator contests were held in the Coliseum. One such contest was the following. There are seven doors; behind four of them are man-eating tigers, and behind the remaining three are doves. The first gladiator in line chooses a door; if there is a tiger behind it, the gladiator is eaten and the tiger is put back behind the door. If there is a dove behind the door, the gladiator is set free, the dove is returned behind its door, and one of the tigers (randomly selected) is killed. If a tiger is killed, there is one less door for the next gladiator to choose from.

The “game” will continue until all the tigers are killed (we assume a potentially infinite supply of gladiators). Run 100 different simulations of this game, indicating at the end of each how many gladiators were eaten and how many were set free during the course of the game. Accumulate statistics overall on the games, so that you can say what the odds are for a gladiator being eaten or set free.

- **11.** Repeat the simulation of the craps game in exercise 4 under the assumption that the dice are “loaded.” For example, assume the following probabilities of a roll on each die of: 1 (0.2), 2 (0.15), 3 (0.15), 4 (0.15), 5 (0.15), 6 (0.2).
- **12.** *Information Theory Simulation Project.* Use the computer to randomly generate simulations of English. The following statistics on a large sample of English prose indicate the relative frequencies of each letter of the alphabet and the space. Figure out a way to make use of this information (it should be entered into your program, perhaps using a DATA statement, and have a parallel array representing the letters of the alphabet each frequency represents) to create a simulated English page of text. Use the random number generator to select which character is chosen each time as the next in the text. How much like English do your simulations look? What ways can you think of to improve the simulation?

FREQUENCIES		
Space (0.1859)	A (0.0642)	B (0.0217)
C (0.0218)	D (0.0317)	E (0.1031)
F (0.0208)	G (0.0152)	H (0.0467)
I (0.0575)	J (0.0008)	K (0.0049)
L (0.0321)	M (0.0198)	N (0.0574)
O (0.0632)	P (0.0152)	Q (0.0008)
R (0.0484)	S (0.0514)	T (0.0796)
U (0.0228)	V (0.0083)	W (0.0175)
X (0.0013)	Y (0.0164)	Z (0.0005)

13. Write a program which will simulate the expected heights of volunteers for the military. The average height is 5'8", and the standard deviation is 4". Since this is a Gaussian, normal distribution, you must use the technique described in this chapter to generate normally distributed values. Plot a histogram of your results, and print out the percentage of candidates you get who are over 6'5" tall.
- 14. Write a program that will generate uniformly distributed random test data lying between any two real values A and B, and write it out to disk. You should probably generate 1000 test points. This could be used to test a program that makes projections regarding levels of precipitation for the environment.
  - 15. Modify problem 14 to generate uniformly distributed *integer* test values between any two specified integer limit values. This could be used to test population-trend programs, for example.
  - 16. Modify problem 14 to generate *normally* distributed random test data, given a mean (*m*) and standard deviation (*s*).
  - 17. Modify problem 14 to generate *exponentially* distributed real data, given a mean (*m*) for the distribution.
  - 18. Write the "Hangman" program described in the section on "Interactive Programs."
  - 19. Write a program which will determine the *center of gravity* of a digitized pattern represented as values stored in a two-dimensional array, as discussed in the "Pattern Recognition" section.

**20.** Write a program which will determine the rectangular area occupied by a pattern, as explained in the text, find its center of gravity (problem 19), and then calculate the sum of "moments" of significant points of the pattern (values exceeding a specified threshold T) normalized by dividing by the area. This is the "signature" of your pattern, and could be compared with the signatures of other patterns.

**21.** *Set exercise.* Define a base set of the names of locations that have been identified as possible sources of oil (the set has 30 members). Then write code that will analyze data (assume it exists on an external medium such as disk) that will determine the sets of these locations that satisfy the following properties: (1) the locations that are within 100 miles of an existing (that is, discovered), source of oil; (2) those locations which have rock formations of class 5 or higher; (3) those locations which have a 60% or greater shale content in the earth; (4) those locations which have nearby mountain ranges (within 5 miles and at least 1000 feet elevation above the plateau level).

Now determine the set which has properties 1, 3, and 5; the set which has properties 2, 4, and 5; the set of all those locations which have *either* property 1, 2, or 4. When you have determined these sets, print out the locations (in characters) which are in the sets. Use logical arrays or packed logical variables, as discussed in the text.

**22.** Write a simulation of a queue which represents arrivals and departures at a busy intersection. Assume that statistically 40% of the time a car gets through the intersection and 60% of the time a new car arrives in a given period of time. The traffic will cause serious "gridlock" in the intersection to the south of this one if the queue of cars gets longer than 20 cars. If a car either arrives or goes through the intersection every 8 seconds, estimate (in 100 simulations) the average length of time until gridlock will occur.

**23.** *The Game of Life.* Invented by the mathematician John H. Conway (and discussed in *Scientific American*, October 1970, p. 120), this game models the growth and changes in a complex collection of living organisms. This model can be interpreted as applying to a collection of microorganisms, an ecologically closed system of plants or animals, or an urban development. Its implications were part of the inspiration for an interesting recent book, *The Recursive Universe*, by William Poundstone (New York: Morrow, 1985).

Start with a clear 50 x 50 board on which "counters" are to be placed. Allow the program user to specify how many counters are to be on the board, and then use a random number generator to place that many counters (if you choose a location that has already been filled, choose again). Remember that there are 2500 locations in your universe, so don't pick too small a population or it will die out rapidly. Each location (except those on the borders) has exactly 8 neighbors. Counters are born, die, or survive during each generation according to the following rules:

*Birth.* Each empty location with exactly three neighbors in its immediately adjacent 8-location neighborhood is a birth location. A counter is placed in this space for the next generation.

*Death.* Counters with 4 or more immediate neighbors die from overcrowding and are removed for the next generation. Counters with zero or one neighbors die of loneliness.

*Survival.* Counters with two or three neighbors survive to the next generation.

“Births” or “deaths” in generation  $n$  do not show up until the next generation ( $n + 1$ ); this means they have no effect on determining the fate of the current generation. The patterns shown indicate the first four generations following a particular initial configuration:

*	***	* *	*	***
***	***		* *	* *
*	***		*	***
		*		

Print out your original board (zeroth generation) and then the next 6–10 generations of that configuration (put in a test not to keep going if everybody died; print out “nobody left” and quit). Use character output. Indicate each generation number.

Assume the edges of the board are “infertile” regions where nothing is born and nothing dies. (Do not place any initial counters in these regions.)

**24.** A battleship sights an unknown enemy vessel 15 nautical miles south of it. The enemy vessel is travelling directly east at 20 knots (a knot is one nautical mile per hour, and a nautical mile is 6076.1 feet). The battleship commences pursuit of the enemy vessel, at a speed of 25 knots; the battleship is always pointing at the escaping vessel (and thus constantly changing direction), from an initial direction of south through various southeast directions. Plot the positions of the two ships at one-minute intervals until the point at which the battleship has caught up with the enemy ship. The motion of the enemy ship is simple—it continues moving east (along the  $x$ -axis), so that its  $y$ -coordinate is always 0 and its  $x$ -coordinate is simply  $x_0$  (which you can consider to be 0) plus its speed times the elapsed time or  $x = v_E t$  (where you must convert the speed into terms of distance per minute). The motion of the battleship is somewhat more complex, since it is constantly changing direction.

The *vector* representing the battleship’s motion is always pointing at the enemy ship; we need to resolve this vector into its  $x$ - and  $y$ - coordinates at each point in time to determine the position of the ship. Its speed is constant, and directed toward the current position of the enemy vessel. Thus, if you imagine a

triangle representing the vector pointing at the enemy vessel as the hypotenuse, the height of the triangle will be the distance of the battleship above the x-axis, that is, its x-coordinate at the last iteration. The base of the triangle will be the distance travelled east by the enemy ship in the time period (that is, the speed in nautical miles per minute times 1 minute), and so will be a constant (call it  $c$ ) you can calculate before the iteration loop. The hypotenuse of this triangle is  $s$ , where  $s = \sqrt{c^2 + y_b^2}$ , where  $y_b$  is the y-coordinate of the battleship at the last iteration. For the battleship,  $v_x = x_b + V_b c/s$ , and  $v_y = y_b - V_b Y_b/s$ . From this information, you should be able to calculate and plot the relative positions of the two ships for each minute until the capture occurs.

- 25.** On a Klingon ship, a subatomic organism is eating the hull of the ship. This organism doubles in size every hour. As an exercise, draw roughly circular areas on a plot which represent the relative size of the organism over the next 6 hours.
- **26.** Using the random number generator, generate a pair of (x, y) coordinates, and then (randomly) create a random walk that begins at that point (this can simulate Brownian motion). Each “step” in the random walk should be limited in size—say, no step should be greater than 5 units in any direction. Try to “plot,” as accurately as possible, the path of the random walk.

**27.** “The self-avoiding random walk.” Models for the folding of polymer molecules, long molecules such as DNA, follow the pattern of the “self-avoiding” random walk. Adapt your random walk model of the previous problem to begin with an empty large array (say, 130 x 130, so that you can print it out), begin with a randomly chosen point (x,y), and then select random new points on the walk, such that they are no more than 7 units in either direction. However, in this case, you must keep track of points on the paths that have already been used, so that the random walk will not cross itself. The best you can do on this model is to block out the best-fitting points to a straight line between your last two positions, and then not allow any new random move to be at any of those filled points. If your random new move hits a filled position, choose again. If you cannot find an unfilled location after 20 tries, then terminate the walk. Run the program for 500 moves in the random walk, or until it terminates after 20 tries to find a new spot, whichever comes first.

Print out your self-avoiding random walk array.

- **28.** Take a 100 x 100 character array, and fill it with randomly chosen points, to try to create a pointillist “painting” (to be displayed at your local art gallery). Set a new seed each time, and experiment with creating 1000 points, then fewer, then more, to see what your best “medium” might be.
- 29.** A “POP” subroutine was written in the text to *pop* a value from the top of a stack, and adjust the stack pointer. Write a PUSH subroutine which will *push* a new value onto an array stack, and adjust the stack pointer.

**30.** *Happy Birthday.* For a category with  $M$  equally probable states, the probability of a second event different from the first is  $(M-1)/M$ ; the probability of a third event different from the first two is  $(M-2)/M$ , . . . , and the probability that the  $I$ th event is different from the previous  $(I-1)$  events is  $(M-I+1)/M$ . Therefore, for  $I$  events, the probability  $P$  that no two have the same value in a group of  $M$  possibilities is:

$$P = \frac{M(M-1)(M-2) \dots (M-I+1)}{M^I}$$

Thus, for example, the probability that no two people in a room of  $N$  people have the same birthday can be calculated from the formula, with  $M = 365$  (ignore leap years). Create a table of the probability that no two people in a room have the same birthday, for room occupancies of 20 to 100. Plot the probability as a function of the room size.

**31.** We read somewhere that a normally distributed random number ( $Y$ ) can be generated from a number  $X$  that comes from a uniform random distribution (such as that created by the RAN function) according to the formula:

$$Y = (X^{0.135} - (1-X)^{0.135})/0.1975$$

Test out this suggestion by generating 1000 numbers according to this formula, using  $X$  values provided by the uniform random number generator on your system, and plot a histogram of the distribution of the values generated. Do they appear normal?

**32.** *Poisson Probabilities.* The Poisson distribution function of the number of occurrences ( $n$ ) during a specified time period, if  $\lambda$ (lambda) is the average number of instances of the event during the given time period, is:

$$P(n) = \frac{\lambda^n e^{-\lambda}}{n!}$$

Use this formula to calculate and plot, as a function of  $n$ , the probability of  $n$  events ( $1 \leq n \leq 50$ ) during a time period of one hour, where the average number ( $\lambda$ ) of events during an hour is 30. Do a similar plot for  $\lambda = 40$  and  $\lambda = 50$ .

- **33.** An “insertion” sort proceeds much in the same way a card player arranges a hand, say, in ascending order. The first (or rightmost) card (or entry) at which your pointer points is taken to be in proper position, and then each card before it is tested to see if it is in proper order; if not, they are interchanged. Write a subroutine to perform an insertion sort.
- **34.** Create a character array  $C(0:100, 0:100)$ , fill it with blanks, and then read in an arbitrary starting location ( $M, N$ ) in the array. Generate 100,000 random values from the RAN generator, and determine a pair of coordinates in the

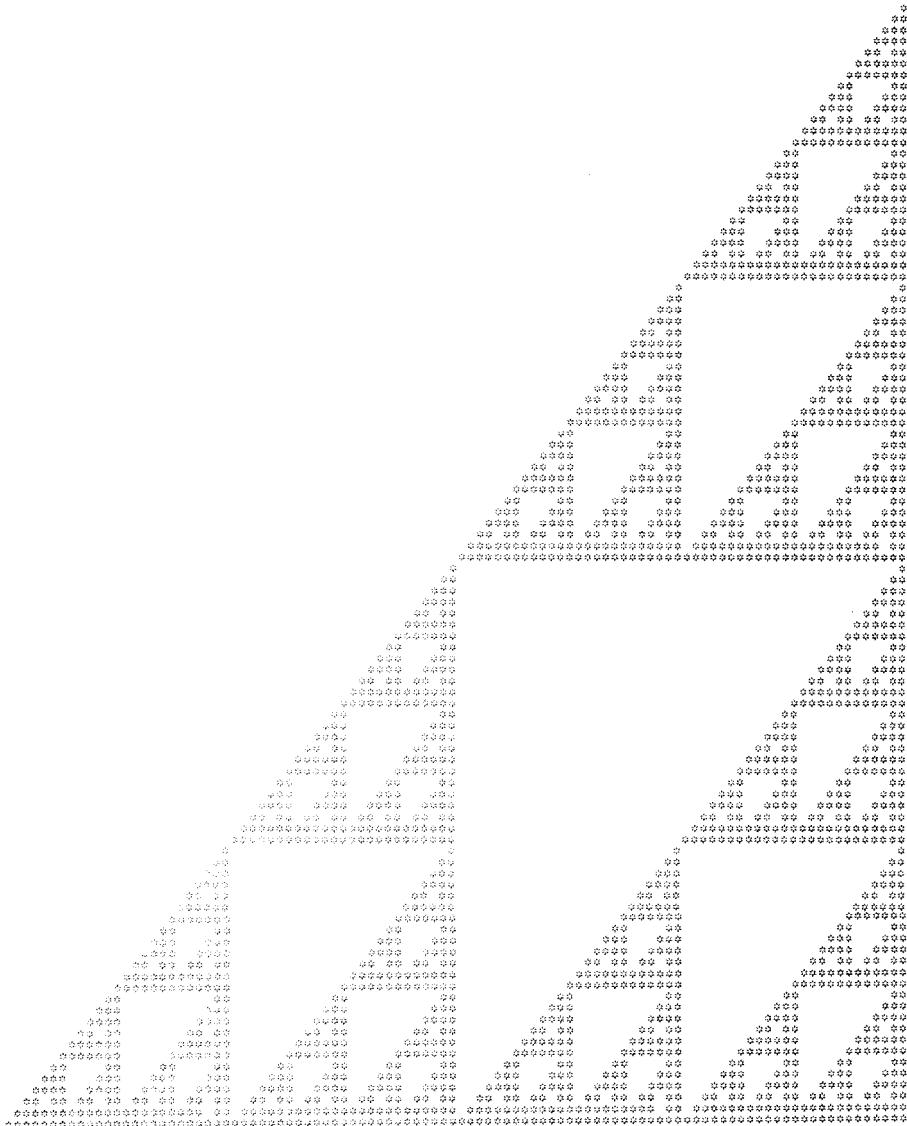
array (M,N) from the previous values, depending on the value from the random number generator (x), as follows:

```

x < 1/3      M = M/2 ;   N = N/2
else if     x < 2/3    M = M/2 ;   N = (N + 100)/2
else          M = (M + 100)/2 ;  N = (N + 100)/2

```

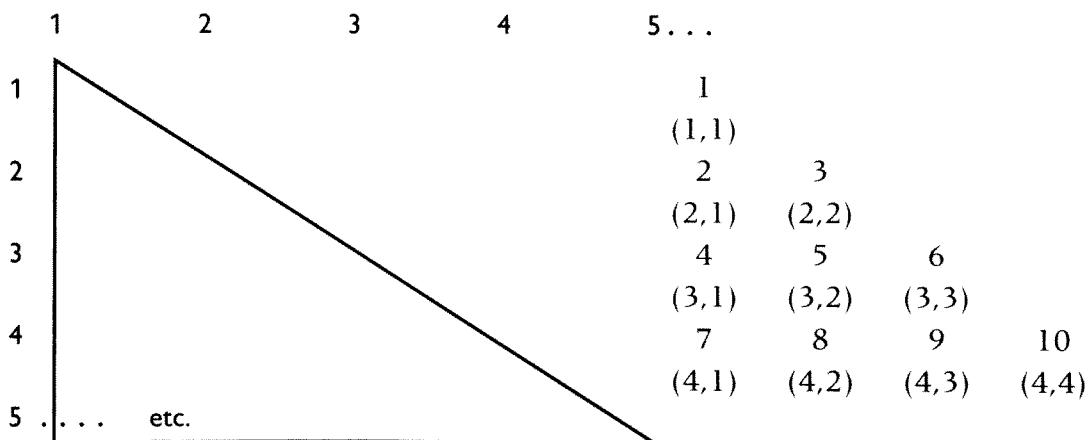
Allow the first 1000 points to be a buildup (that is, points that are not counted); after that, store a point character (such as '\*') in the array at position (M,N). At the end, print out the array, beginning at row 100 and working down to row 0, a row at a time, across the page. It should look like the following (and you should note that the starting point (M,N) does not make any difference in the end result). This is a fractal pattern:



**35.** "Once in a Blue Moon." This phrase suggests that such an event does not occur very often. Let us run a simulation to get an idea just *how* often it occurs. A *blue moon* is said to occur if two full moons fall in the same month. The most recent "blue moon" happened in December, 1990; there were full moons on both December 2 and December 31, 1990. The length of a *lunar month* is 29 days, 12 hours, 44 minutes, and 2.7+ seconds. Let us assume that the moon became full on December 31, 1990, at 8 p.m. EST. Given our knowledge of calendar dates developed in earlier exercises, determine when the *next* blue moon will occur. Then run a simulation for the next 1000 years, counting occurrences of blue moons, to arrive at a rough estimate of how often (on the average) they occur.

**36.** *Directed Graphs.* A directed graph (digraph) is a set of relations among objects (nodes or vertices) which may or may not be connected (by edges). An edge may be one-way (that is, connecting object A to object B, but not vice versa), or two-way, or may not exist at all. An edge value may simply be represented by a 1 (or 0 if it does not exist), or it may be weighted (to represent distance, or likelihood, etc.). A two-dimensional matrix ( $n \times n$ ) may be used to represent the digraph relations among  $n$  objects (let us say, for the moment, the matrix will contain 1s where connections exist, 0s elsewhere). Write a program to read in a 2-D matrix (of some reasonable size  $n \times n$ , say  $2 < n \leq 12$ ), and plot the  $n$  objects with the various edges connecting them. Since such a directed graph could also represent distances between points on a map, write a program which will accept such a digraph matrix and determine all the possible routes (if any) between two specified nodes ( $n$  small).

**37.** Storing a *lower triangular matrix*. Such a matrix has all zero values above the major diagonal. It can be mapped onto a one-dimensional array using the following algorithm: if the value in the lower triangle is in location  $(r,c)$ , then store it into location  $(r*(r-1)/2 + c)$  in the one-dimensional array. Write efficient code to transfer the data from the triangular array to its new 1-dimensional home.



**38.** *Paychecks.* Write a FORTRAN program which will read in a paycheck amount of the form \$xxxx.xx (or use your random number generator to create paycheck values in that range), plus the name of the person to whom the check is to be paid, and will write out the check (nicely formatted), including the amount of dollars written out in words, as is normally required on a check. You might find the following reference array useful:

```
CHARACTER CASH(27)*10
DATA CASH/'ONE', 'TWO', 'THREE', 'FOUR', 'FIVE', 'SIX', ... 'TEN',
&'ELEVEN', 'TWELVE', . . . 'NINETEEN', 'TWENTY', 'THIRTY',
&'FORTY', 'FIFTY', 'SIXTY', 'SEVENTY', 'EIGHTY', 'NINETY'/

```

**39.** *Spell-Checker.* Write a program that stores a “dictionary” of reference words and checks an input text against the reference array for misspelled words (this means that any word you use in the text will have to be in the dictionary, or it will be flagged as misspelled; thus you will have to have a large dictionary or a very simple text). Try to be clever in the way you handle plurals, past tenses of verbs, etc., and try to be efficient in the way you store words in the dictionary for reference, so that each word in the text does not have to be checked against *every* word in the dictionary. Such spell-checkers are used today by many people (including scientists) who write reports, letters, and the like on a computer.

**40.** *Readability Index.* There are various measures proposed for the “readability” of a text, and the U.S. Government (Department of Defense) Military Standard MIL-M-38784B requires the use of such a measure by contractors who produce manuals for the armed services. The Flesch-Kincaid measure used in this document is the following: to calculate the Overall Reading Grade Level (OGL) of a text, determine the average sentence length (ASL) of the text (number of words / number of sentences) and the average number of syllables per word (ASW = number of syllables / number of words), and then

$$\text{OGL} = .39 * \text{ASL} + 11.8 * \text{ASW} - 15.59$$

(round to nearest tenth).

A good range is considered to be 6–10, and the recommended level for Naval Electronics and Sea System Commands is 9th grade. Write a program that will read in a text and evaluate its OGL. Probably the best way to determine syllables is by counting the number of vowels or vowel pairs in a word.

**41.** Write an “operation” function which will combine two chemical compounds, as discussed in the section on Fortran 90 derived data types. If you have a Fortran 90 compiler available, try it out.



## SUGGESTED READINGS

- Graybeal, Wayne T., and Udo W. Pooch. *Simulation: Principles and Methods*. Boston: Little, Brown, 1980.
- Holmes, Willard M. (ed.). *Artificial Intelligence and Simulation*. San Diego: Society for Computer Simulation, 1985.
- Kac, Mark. "What is Random?", *American Scientist*, 71 (1983), 405–6.
- Kalos, Malvin H., and Paula A. Whitlock. *Monte Carlo Methods. Vol. I: Basics*. New York: Wiley, 1986.
- Lehman, Richard S. *Computer Simulation and Modeling: An Introduction*. New York: Lawrence Erlbaum, 1977.
- Metropolis, Nicholas, and Stanislaw Ulam. "The Monte Carlo Method." *Journal of the American Statistical Association*, 44, no. 247 (September, 1949), 335–341.
- Payne, James A. *Introduction to Simulation: Programming Techniques and Methods of Analysis*. New York: McGraw-Hill, 1982.
- Poundstone, William. *The Recursive Universe: Cosmic Complexity and the Limits of Scientific Knowledge*. New York: Morrow, 1985.
- Solomon, Susan L. *Simulation of Waiting-Line Systems*. Englewood Cliffs, N. J.: Prentice-Hall, 1983.
- Ulam, Stanislaw M. *Adventures of a Mathematician*. New York: Charles Scribner's Sons, 1976.
- Von Neumann, John, and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton: Princeton University Press, 1943.

## Neural Nets

- Denker, John S. (ed.). *Neural Networks for Computing*, AIP Conference Proceedings 151. New York: American Institute of Physics, 1986.
- Firebaugh, Morris W. *Artificial Intelligence: A Knowledge-Based Approach*. Boston: Boyd & Fraser, 1988.
- Graubard, Stephen R. (ed.). *The Artificial Intelligence Debate*. Cambridge, Mass.: MIT Press, 1988.
- Hopfield, John J. "Neural Networks and Physical Systems with Emergency Collective Computational Abilities," in *Proceedings of the National Academy of Sciences* 79 (1982), 2554.

Nelson, Marilyn McCord, and W. T. Illingworth. *A Practical Guide to Neural Nets*, Reading, Mass.: Addison-Wesley, 1991.

## Object Orientation

Cox, Brad J. *Object Oriented Programming: An Evolutionary Approach*. Reading, Mass.: Addison-Wesley, 1987.

Khoshafian, Setrag, and Razmik Abnous. *Object Orientation: Concepts, Languages, Databases, User Interfaces*. New York: Wiley, 1990.

# CHAPTER 14



# NUMERIC METHODS

Powerful numeric techniques can be implemented on the computer using FORTRAN. The number of repetitions needed for accuracy in the solution to many problems are impossible for a human, but are quite feasible given the use of a computer. Some of the applications covered include root finding, determining a best-fitting equation to describe a set of data, methods to improve various kinds of sensed data, integration, solving simultaneous equations, and various other matrix manipulations. These are all valuable tools for the practicing scientist.

Otherwise obscure details of the surface of Enceladus, a moon of Saturn, are visible because of the computerized process of photographic image enhancement.

*"Though this be madness, yet there is method in it."*

- William Shakespeare, Hamlet II, ii, 211

A group of powerful techniques for solving various numeric problems in the sciences lend themselves to implementation in computer routines. We will briefly examine a few of the most commonly encountered of these techniques in this chapter, to give the reader a taste of these methods, their applications, and the programming methods involved.

"The purpose of computing is insight, not numbers." - R. Hamming



## ROOT FINDING

Situations may arise where we need to find the *roots* of an equation. If the equation is of a simple linear form, such as

$$ax = b$$

the solution is just as simple, that is,  $x = b/a$ . Even if the equation is in the form of a quadratic, such as:

$$ax^2 + bx + c = 0 \quad (a \neq 0)$$

we know the pair of roots for this equation.

However, not all equations for which we wish to find roots are so simple. We must look into various methods for finding the roots, or "zeros," of any equation  $f(x)$  so that:

$$f(x) = 0$$

If we were to graph such a function, the points at which it crosses the x-axis are the solutions we are looking for. Some solutions can be arrived at by inspecting a graph. Others take the general form of making a guess (and, if at all possible, it should be a good guess) at the value of the root, and then *iterating*, or homing in, to a solution. Our termination criteria for having arrived at a good approximation to the root may be:

$$|x' - x| < \text{epsilon} \quad \text{or} \quad |f(x')| < \text{epsilon}$$

The first criterion is familiar from our work with approximating functions defined in terms of infinite series.

*Simple iteration* is feasible only if you can express the equation readily in some form where you can isolate the unknown  $x$  on the left side of a version of the equation. If you can rearrange the equation this way, this is the form you will give to your iterative program. Begin by making a guess at the value

of the root. Use your first guess in the right side of the equation to obtain the next approximation, and so on.

Each new approximation will then be obtained by plugging the last approximation into the right side of the equation. This process should continue until our criteria for termination are met.

The *bisection method* relies on determining two points (A, B) at which the values of the function we are examining are of different signs; we then know that a zero (or root) of the function will lie between these two values if the function is continuous. Beginning with these two points, we can halve the difference between them and examine the value of the function at that point (C). F(C) will have the same sign as one of the two original points, and the zero will lie between this new midpoint and the point at which the function has the opposite sign. We can then halve the difference between these two values and repeat the procedure. We continue repeating this operation until the value of the function at the last midpoint is within our tolerance level epsilon.

```
***** BISECTION ROOT FUNCTION *****
FUNCTION BISECT(FUN, POS, NEG)
REAL NEG, MID, BISECT, EPSI, FUNN, FUNP, FUNM
INTEGER LIMIT, ITERS
***** SET LIMIT TO NUMBER OF ITERATIONS *****
***** AND TOLERANCE LEVEL FOR COUNTING A SOLUTION *****
DATA LIMIT, EPSI / 50, 0.00001 /
FUNP = FUN(POS)
FUNN = FUN(NEG)
IF (FUNP*FUNN.GT.0) THEN
    PRINT*, 'YOU MUST ENTER TWO VALUES AT WHICH SIGN CHANGES'
    BISECT = -99999999.99
    RETURN
ENDIF
ITERS = 0
5 CONTINUE
MID = (POS+NEG)/2
FUNM = FUN(MID)
IF(ABS(FUNM).LE.EPSI) THEN
    BISECT = MID
    RETURN
ENDIF
***** DETERMINE WHICH HALF THE ROOT LIES IN NOW *****
IF(FUNM*FUNP.LT.0) THEN
***** IT IS BETWEEN MID AND POS *****
    NEG = MID
ELSE
    POS = MID
```

```

        FUNP = FUNM
ENDIF
ITERS = ITERS + 1
IF(ITERS.LE.LIMIT) GO TO 5
BISECT = MID
RETURN
END

```

The *Newton, or Newton-Raphson, method* uses the *tangent* to the curve to find the next approximation to the root (the point  $x_0$  where the tangent crosses the x-axis).

A difficulty, from a user's point of view, is that formulas for both the function and its derivative must be supplied to the N-R routine, since computer procedures for finding the derivative form of a function are awkward and time-consuming at best. We will assume in the example that the user has provided both the function whose root is sought (FUN) and the form of the function's derivative (DFUN). These would have to have been declared as INTRINSIC or EXTERNAL in the program unit that references the APPLE function.

```

***** NEWTON-RAPHSON ROOT FINDER *****
FUNCTION APPLE (FUN, DFUN, A)
REAL APPLE, FUN, DFUN, A, X, EPSI, FUNX
X = A
EPSI = .00001
FUNX = FUN(X)
6 CONTINUE
DAPPLE = DFUN(X)
***** MAKE SURE DIVISOR ISN'T TOO CLOSE TO ZERO *****
IF(ABS(DAPPLE).GE. 0.0001) THEN
    X = X - FUNX/DAPPLE
    FUNX = FUN(X)
ENDIF
IF (ABS(FUNX).GT.EPSI) GO TO 6
APPLE = X
RETURN
END

```

This is the fastest method we have examined, but it may not converge. The bisection method is the safest simple method, but it is rather slow.

 DATA IMPROVEMENT

## Smoothing Data

Often the data obtained from experiments are error-prone. One way to minimize the effects of such errors is by *smoothing* the data recorded. One common method for such smoothing is to replace each data element with the average of the three data points of which it is the middle value.

## Enhancing Data

A related topic is a correlate of the data-smoothing technique. The need may arise to reduce noise and emphasize contrasts in data that is to be processed/analyzed/classified. A major application area for such techniques is that of *remote sensing*, which includes the processing of varieties of data (photographic, radiometric, etc.) acquired by orbiting satellites. Such data may need to be magnified, scaled down, cleared of noise, enhanced, or categorized. A matrix of data (most likely a digitized photographic image) may need to be *reduced*. This means taking a large matrix and reducing it to a smaller one for output. One algorithm is, to reduce the picture by a factor of  $m^2$ , simply select every  $m$ th row and every  $m$ th column for the output array. Each entry in the array represents a pixel (picture element) and its related level of brightness. Another approach might be to *average* the brightness levels in every  $m \times m$  square and use that average in the reduced picture.

An image can also be magnified; this is known as *zooming*. To do this, a smaller array is reproduced in a larger one. If the magnification is to be by  $m^2$ , then each pixel in a particular row and column location is blown up into an  $m \times m$  block of entries for the larger array, all containing the same brightness value as the original pixel.

Images may be *rectified* and *restored* by systematically accounting for known distortions, such as that caused by the rotation of the earth. This is done by applying transformation functions, derived by those working in the area, to determine the values for the undistorted matrix. A significant problem in remote sensing is that of noise in the data, which may be either periodic or random. For example, random noise tends to produce "spikes," where the noise values change more rapidly than surrounding values. Spikes can be recognized by looking at an average of the pixel values in a neighborhood. If there is a difference exceeding a specified threshold of a pixel from its neighbors, it is recognized as a noisy entry, and is replaced by the average of the values in its neighborhood. A fairly standard window size for such analysis is  $3 \times 3$  or  $5 \times 5$ .

Images may be *enhanced* to improve readability by the human eye. This is done after as much noise has been removed as possible. Enhancements may be *global*, modifying brightness values throughout an image according to some criterion, or they may be *local*, modifying values based on those of neighboring pixels. An example of a global enhancement would be one which simply expanded the brightness range of the image to fill the range available on the output device, such as a CRT. The typical range of different output values available on such a device is from 0 to 255. A particular image probably will not have brightness values covering that whole spectrum. One can determine the MIN and MAX brightness values in the image, and then alter each pixel value in the image according to the following "linear stretching" equation:

$$N' = ((N - \text{MIN}) / (\text{MAX} - \text{MIN})) * 255$$

where N is the original number representing the brightness level of a pixel, and N' is the enhanced value. Such a stretching algorithm may also be weighted, to enhance the more frequently occurring brightness levels (determined from a histogram) more, and the less frequently occurring levels would be assigned to be stretched over a narrower range of the remaining spectrum. Thus more significant (in the sense of more frequently occurring) values would be more greatly enhanced.

A threshold, or mask, may be applied to enhance the contrast between two gray levels, such as those representing land and water. An empirical threshold can be set which succeeds in separating the two classes, and then the one falling below the threshold (say, land) can be set to black (0) to enhance the appearance of the water brightness contrasts.

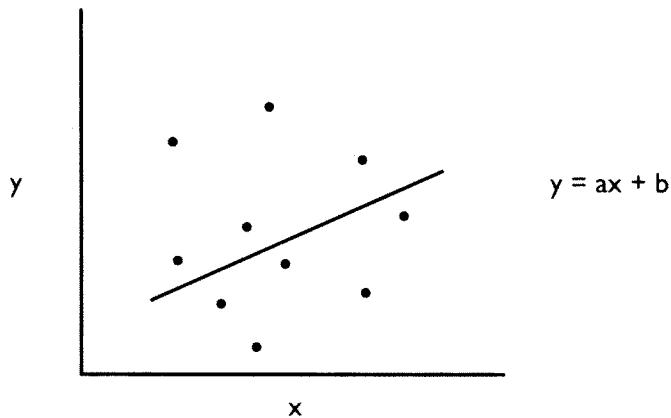
*Filtering* can stress or de-emphasize high or low frequency features. This can be done by passing a window (normally 3 x 3, 5 x 5, or 7 x 7) across the image and replacing the central value in the window by the average of the values of its neighbors. If the window also assigns weights to each of the elements it covers, the process is called *convolution* and the central pixel of the window receives the weighted average value of its neighbors.

The techniques we have discussed have general utility in other areas of pattern recognition as well. The interested reader is referred to the books on remote sensing listed at the end of this chapter.



## LEAST-SQUARES FITTING OF DATA

If we assume our data contains some inherent error, we may attempt to minimize it by looking for some pattern, such as a straight line, which the data seems generally to fit. Let us assume some set of data like that shown, for which we wish to find the "best-fitting" line or curve. The equation of a straight line passing among the points is:



The "trick" is to find the coefficients  $a$  and  $b$  such that the line lies as closely as possible in the midst of the points, not at too great a distance from any of them. To do this, we have to minimize the (let us choose vertical) distance, on the average, of points from the line. Since we do not want a point a distance of  $-c$  below the line to "balance" a point a distance of  $+c$  above the line and thus say that their average distance from the line is 0, we should look at the squares of the  $y$  distances from the line. For a best-fitting straight line to the points, we want the average of these squares of distances to be as small as possible (which gives rise to the term "least squares").

Suppose we have a set of  $n$  points to fit, with coordinates  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ . For a given data point  $(x_i, y_i)$ , the corresponding point *on* the line will be the point at  $(x_i, y = ax_i + b)$ . Thus each distance is  $(y_i - ax_i - b)$ , and it is the sum of the squares of these distances that must be minimized. To minimize this function,

$$f(a, b) = \sum (ax_i + b - y_i)^2$$

we take the partial derivatives with respect to  $a$  and  $b$ , respectively, and set them to 0, thus obtaining the following values for  $a$  and  $b$ :

$$\delta f / \delta a = \sum 2(ax_i + b - y_i)x_i = 0; \quad \left( \sum x_i^2 \right)a + \left( \sum x_i \right)b - \sum x_i y_i = 0$$

$$\delta f / \delta b = \sum 2(ax_i + b - y_i) = 0; \quad \left( \sum x_i \right)a + nb - \sum y_i = 0$$

and from these we obtain:

$$a = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - (\sum x_i)^2}$$

$$b = \frac{\sum y_i - a \sum x_i}{n}$$

Or, if the averages  $\bar{Y}$  and  $\bar{X}$  have been calculated for the set of  $n$  points, then these can be expressed more simply in these terms:

$$a = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2} \quad b = \bar{Y} - a\bar{X}$$

where  $X = \sum x_i/n$  and  $Y = \sum y_i/n$ .

Similarly, we could fit an exponential curve to a set of data points, with an equation of the curve such as

$$y = cx^d$$

that is,

$$\log y = \log c + d \log x = d \log x + \log c$$

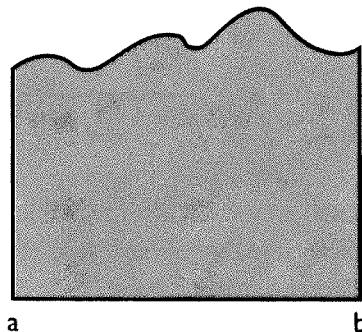
We can treat this equation just as we did the one for the straight line, except this time we minimize variations in  $\log y$ .

In either case, these coefficients give us a regression line or equation, from which we can predict the values of new data points (given  $y$ , to predict  $x$ , or vice versa).



## NUMERIC INTEGRATION

Many technical problems involve the calculation of the area under a curve, such as the one shown. If the curve is represented as  $f(x)$ , then the area under the curve between points  $a$  and  $b$  on the  $x$ -axis is the definite integral:

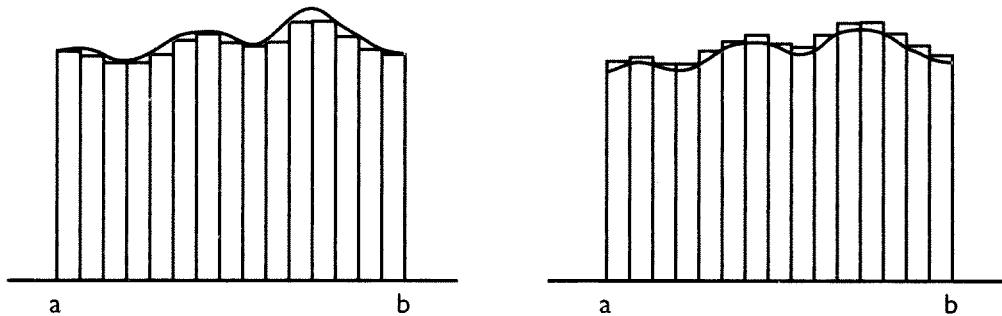


$$\text{area} = \int_a^b f(x) dx$$

There are various methods of determining such areas.

## Rectangular Quadrature

For a great many functions, calculus does not provide us with a neat analytical solution. Thus we are left with the problem of approximating the area under a curve described by such a function and must resort to other methods to arrive at a value. A crude approach is to plot the curve on graph paper ruled into squares and then try to count the number of squares that lie under the curve. We can implement this technique on the computer by fitting a set of rectangles (the more the better) under the curve. Such a sum will clearly be less than the area we want. We could also compute the area of rectangles which *enclose* the curve, and even average the two sums to come up with an approximation.



We can divide the range from  $a$  to  $b$  into  $n$  equal segments, each of width  $(b-a)/n$ . The area *under* the curve, approximated by the rectangles in the left picture, can then be calculated by multiplying the rectangle width  $((b-a)/n)$  by the minimum value of the curve in the range covered by the rectangle, and the *enclosing* area (right picture) can be calculated by multiplying the rectangle width by the maximum value of the function in the range of the rectangle. We may compromise by calculating the value of the area of the rectangle whose height is the value of the function at the *midpoint* of the rectangle.

Such an area calculation, or *quadrature*, could be determined by the following FORTRAN function, which accepts the function, the endpoints  $a$  and  $b$ , and a value for  $n$ , the number of rectangles to be used. Thus we can increase or decrease  $n$  as time or patience permit.

```
***** FUNCTION TO CALCULATE AREA UNDER A CURVE *****
***** USING RECTANGLES WHOSE HEIGHT IS THE MIDPOINT VALUE *****
***** OF THE FUNCTION *****
FUNCTION RAREA(FUN, A, B, N)
***** FUN IS NAME OF FUNCTION TO BE INTEGRATED *****
***** A IS LEFT ENDPOINT FOR INTEGRAL, B IS RIGHT ENDPOINT *****
***** N IS THE NUMBER OF RECTANGLES USED *****
INTEGER N
REAL FUN, RAREA, A, B, WIDTH, SUM, X
```

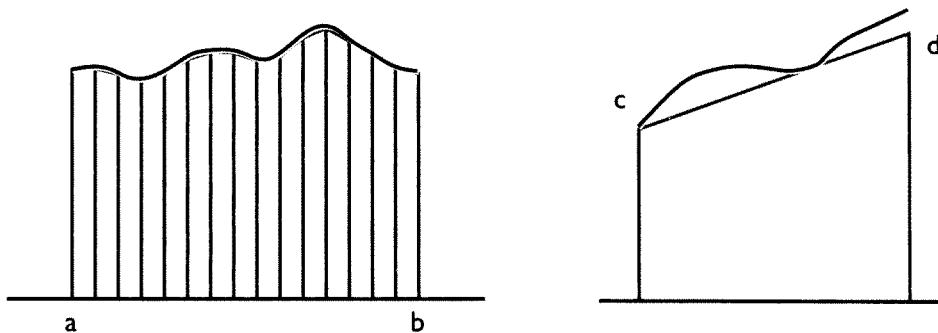
```

WIDTH = (B-A)/N
***** FIRST MIDPOINT OCCURS AT A + WIDTH/2 *****
X = A + WIDTH/2
SUM = 0
DO 8 I = 1, N
    SUM = SUM + FUN(X)
    X = X + WIDTH
8 CONTINUE
***** MULTIPLIER FACTORED OUT TO SAVE OPERATIONS NEEDED *****
SUM = SUM*WIDTH
RAREA = SUM
RETURN
END

```

## Trapezoidal Rule

Our rectangle quadrature for the area under the curve defined by a function does not give us the best estimate. We can improve upon this procedure by fitting trapezoids instead of rectangles to the curve. If we use a sufficiently large number of trapezoids, the straight line "top" of each trapezoid should be a relatively good approximation to the curve itself. The area of a trapezoid of width  $dx$  with heights  $c$  and  $d$  is  $(dx)((c+d)/2)$ . We apply this approach in the following function to find an area under any specified function  $FUN$  in the range from  $A$  to  $B$ , using  $N$  trapezoids. If time permits, increasing  $N$  will increase the accuracy of our estimate of the area.



```

***** FUNCTION TO CALCULATE AREA USING TRAPEZOIDAL RULE *****
FUNCTION TRAP(FUN, A, B, N)
***** FUNCTION FUN IS PASSED AS EXTERNAL OR INTRINSIC *****
***** A IS LEFT ENDPOINT OF INTEGRAL, B IS RIGHT ENDPOINT *****
***** N IS SPECIFIED NUMBER OF TRAPEZIODS USED *****
REAL LEFT, TRAP, FUN, A, B, WIDTH, RIGHT, HEIGHT, SUM
INTEGER N
WIDTH = (B-A)/N

```

```

LEFT = A
SUM = 0
DO 40 I = 1, N
    RIGHT = LEFT + WIDTH
    HEIGHT = (FUN(LEFT)+FUN(RIGHT))/2
    SUM = SUM + WIDTH*HEIGHT
    LEFT = RIGHT
40 CONTINUE
TRAP = SUM
RETURN
END

```

This function can be made more efficient by taking advantage of the fact that the right height ( $d$ ) of each inner trapezoid is equal to the left height ( $c$ ) of the next trapezoid. Thus for two adjacent trapezoids, we get the two heights represented as:

$$\frac{f(c) + f(d)}{2} + \frac{f(d) + f(e)}{2} = \frac{f(c)}{2} + f(d) + \frac{f(e)}{2}$$

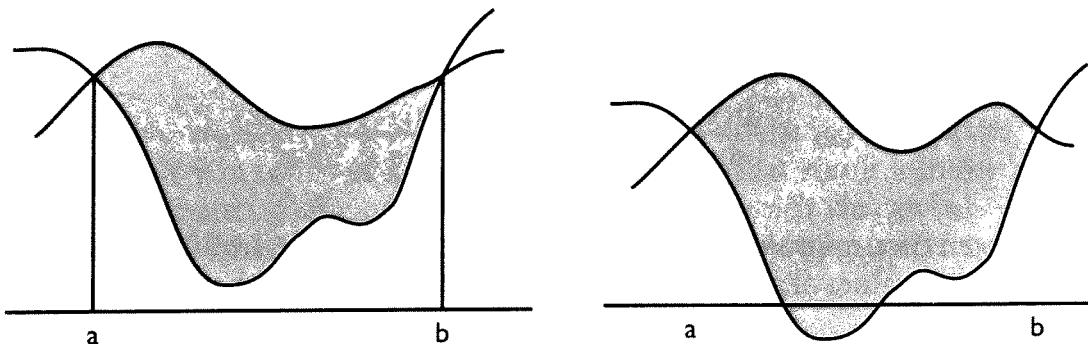
Each height (function value at one of the interior points), except for those at the endpoints, appears exactly once, which greatly simplifies the calculation. Furthermore, the same width is used for each trapezoid, so we can change our program to:

```

SUM = (FUN(A) + FUN(B))/2
DO 40 I = 2, N - 1
    LEFT = LEFT + WIDTH
    SUM = SUM + WIDTH*FUN(LEFT)
40 CONTINUE

```

For areas *between* two curves, instead of between a curve and the x-axis, if the curves cross at points  $a$  and  $b$ , we would only have to calculate the area between the upper curve ( $f(x)$ ) and the x-axis, according to one of our methods, and then subtract the area between the lower curve ( $g(x)$ ) and the x-axis. A little thought will reveal that this can be accomplished by applying one of our quadrature techniques to a function of the form ( $f(x) - g(x)$ ). This works even if part of the area in question drops below the x-axis, as in the right diagram.



In physical applications, integration may be used to calculate total work done when a given force  $F(x)$  is applied:

$$\int_a^b F(x) \, dx$$

and if the force has a simple form, such as the restoring force on a particle attached to a stretched spring ( $-kx$ ), it may easily be determined from the calculus formulation of the result of the integration. However, in many cases, the expression for the force may not be one easily integrable by normal methods, and we may have to apply our numeric quadratures. Integrals are also used to calculate centers of gravity of objects, volumes of odd-shaped objects, hydrostatic pressure in a container, and so on.



## DIFFERENTIATION

There are many functions for which calculus provides us with the form of the derivative of the function. However, in a case where we have a table of values rather than an analytic function, or a function that is too difficult to differentiate, we may need to apply numeric methods to find the derivative.

If we have a recalcitrant function, which does not readily lend itself to the methods of the calculus for taking derivatives, we can then make use of the definition of the derivative in terms of a limit:

$$f'(x) = \frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} = \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

This could be altered to look at the limit of the expression approached from two directions,  $(+\Delta x)$  and  $(-\Delta x)$ . The formula then becomes:

$$f'(x) = \frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x}$$

If we let  $a = 2\Delta x$ , we can rewrite this derivative as:

$$f'(x) = \frac{df(x)}{dx} = \lim_{a \rightarrow 0} = \frac{f(x + a/2) - f(x - a/2)}{a}$$

We can then approximate the derivative by calculating the limit for smaller and smaller values of  $a$  (though not letting it get *too* small, so that we are in dan-

ger of dividing by zero). We can also apply the limit definition to the previous formula to get a formula for the second derivative:

$$f''(x) = \frac{df'(x)}{dx} = \frac{f'(x + a/2) - f'(x - a/2)}{a}$$

and substituting in this equation

$$\begin{aligned}f'(x + a/2) &= [f(x + a) - f(x)]/a \\f'(x - a/2) &= [f(x) - f(x - a)]/a\end{aligned}$$

we get:

$$f''(x) = [f(x + a) - 2f(x) + f(x - a)]/a^2$$

For either of these equations, we can plug in a value for the point  $x$  at which we want to calculate the derivative, and take a small value for  $a$ , to come up with an approximate value. You can see that higher-order derivatives may be calculated in a similar manner. Derivatives are used to calculate tangents to curves, values for accelerations, to calculate marginal cost in economics, to aid in plotting curves, and other applications.



## SOLVING SYSTEMS OF SIMULTANEOUS LINEAR EQUATIONS

A linear equation is one in which no powers of the unknown variable (higher than 1) appear. For instance,

$$3x = 12$$

is a linear equation in one unknown,  $x$ . Another simple system is

$$\begin{aligned}2x - 3y &= 1 \\3x - 2y &= 4\end{aligned}$$

which is a pair of linear equations in two unknowns,  $x$  and  $y$ . This can easily be solved, as:

$$\begin{aligned}x &= [-2(1) + 3(4)]/[2(-2) + 3(3)] = 2 \\y &= [2(4) - 3(1)]/[2(-2) + 3(3)] = 1\end{aligned}$$

We have worked with such equations in secondary school, solving one equation for  $y$  in terms of  $x$ , and then substituting that result into the other equation, obtaining an equation in one unknown, which we can solve for  $x$ . Once  $x$  is known, we can solve for  $y$ . In general, if our two equations have the form:

$$\begin{aligned} a_1 x + b_1 y &= c_1 \\ a_2 x + b_2 y &= c_2 \end{aligned}$$

if the system is *consistent*, then the solution is of the form:

$$\begin{aligned} x &= (b_2 c_1 - b_1 c_2) / (a_1 b_2 - a_2 b_1) \\ y &= (a_1 c_2 - a_2 c_1) / (a_1 b_2 - a_2 b_1) \end{aligned}$$

In general, a system of  $n$  equations in  $n$  unknowns can be written in the following form:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \dots + a_{1n}x_n &= c_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \dots + a_{2n}x_n &= c_2 \\ \dots \dots \dots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \dots + a_{nn}x_n &= c_n \end{aligned}$$

The double subscripting on the coefficients is very familiar to us from our use of two-dimensional arrays, and we will use such an array (a matrix) to solve such a set of equations. When the order of the set of equations gets greater than 2, it becomes difficult to solve the equation set by hand, and it is a good time to bring in some powerful numeric methods which will allow us to have the computer find the solutions for us.

Another way of writing our system of equations is as a matrix product  $AX = C$ , where  $A$  represents the matrix of coefficients:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$$

## Gaussian Elimination

First we will create an *augmented* matrix from the matrix form of the equation given, by adding the constant vector  $C$  to the coefficient matrix  $A$  as its  $n+1$ st column. Thus our augmented matrix will look as follows:

$$\left| \begin{array}{cccc|c} a_{11} & a_{12} & a_{13} & \dots & a_{1n} & c_1 \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} & c_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} & c_n \end{array} \right|$$

We noticed in our earlier discussion that the method we learned for solving a simple set of equations in more than one unknown was to solve for one of the unknowns in terms of the other(s). This formulation is then substituted

back into an earlier equation, until we can get one equation in one unknown, which we know how to solve. Generally, this sort of operation on a large set of equations utilizes these rules:

- Multiply an equation by a nonzero constant.
- Interchange two equations.
- Add a multiple of one equation to another equation.

In our matrix notation, substitute "row" for "equation" in these rules. Our goal is that the coefficient matrix A be transformed into an "upper triangular" matrix, which has all zero values below the major diagonal. In this way, it will have an explicit solution for  $x_n$  which can then be substituted in row  $n-1$  to obtain a solution for  $x_{n-1}$ . These two values can then be substituted into row  $n-2$  to obtain a value for  $x_{n-2}$ , and so on.

One method is to begin by making the coefficients in the first column all 1s. This can readily be done by "normalizing" each row by dividing through by the element in the first column of that row (unless it is zero). Since our purpose will eventually be to eliminate all of the first column coefficients except in row 1, if the coefficient in the first column is already zero, we will go on to the next row. If the first column value is zero for row 1, we will merely interchange that row with the first row that we find with a nonzero first column element. This logic can be easily implemented in a simple subroutine which will make the first column coefficients all 1 (or 0) for any real-valued  $n \times m$  matrix.

```
***** NORMALIZE ALL FIRST COLUMN COEFFICIENTS TO 1 *****
SUBROUTINE NORM (A, N, M)
INTEGER N, M, I, J
REAL A(N, M), TEMP, DIV
***** FIRST ROW SHOULD HAVE NON-ZERO FIRST COLUMN *****
IF (A(1,1).EQ.0) THEN
    I = 2
3   IF(A(I,1).NE.0) THEN
        DO 4 J = 1, M
            TEMP = A(1,J)
            A(1,J) = A(I, J)
            A(I, J) = TEMP
4   CONTINUE
ELSE
    I = I + 1
    GO TO 3
ENDIF
ENDIF
DO 9 I = 1, N
    IF (A(I,1).NE.0) THEN
```

```

DIV = A(I,1)
DO 8 J = 1, M
8      A(I,J) = A(I,J)/DIV
      ENDIF
9  CONTINUE
      RETURN
END

```

The first row can then be subtracted from each of the rows below it. This will leave us with a matrix of the form:

$$\begin{bmatrix} 1 & a'_{12} & a'_{13} & \dots & a'_{1n} & c'_1 \\ 0 & a'_{22} & a'_{23} & \dots & a'_{2n} & c'_2 \\ \dots \\ 0 & a'_{n2} & a'_{n3} & \dots & a'_{nn} & c'_n \end{bmatrix}$$

The continuation of the method should now be fairly obvious. We then normalize all second column entries, beginning with the second row, to 1 (unless they are already zero). If the second column entry in the second row is 0, we interchange it with the first row below it with a non-zero second column entry. One could readily generalize the normalization subroutine we wrote earlier to normalize the  $k$ th column entry to 1 for all rows from row  $k$  to the end of the matrix. It would be best before beginning this procedure to be sure that each row  $j$  had a nonzero value in its  $j$ th column (that is, a nonzero diagonal).

After all of the normalizations and row subtractions are completed, we are left with an upper triangular matrix, as discussed, with all 1s on the diagonal. By "back-substitution," we can now readily solve for each of the unknown  $x$  values, beginning with the  $n$ th value. Our modified matrix representation of the equations  $A'X = C'$  will look as follows:

$$\begin{bmatrix} 1 & a'_{12} & a'_{13} & \dots & a'_{1n} \\ 0 & 1 & a'_{23} & \dots & a'_{2n} \\ 0 & 0 & 1 & \dots & a'_{3n} \\ \dots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} c'_1 \\ c'_2 \\ c'_3 \\ \vdots \\ c'_n \end{bmatrix}$$

from which we can readily see that  $x_n$  will equal  $c'_n$ ,  $x_{n-1}$  will equal  $c'_{n-1} - a'_{n-1,n}x_n$ , and so forth. Since the  $c'$  values are actually still values in the augmented, modified  $A$  array, we can implement the back-substitution in the following subroutine.

```
***** BACK-SUBSTITUTION FOR GAUSSIAN ELIMINATION *****
SUBROUTINE BACK (A, N, M, X)
INTEGER M, N, I, K
REAL A(N,M), X(N), SUM
DO 20 I = N, 1, -1
***** BEGIN SUM TERM WITH C' (CONSTANT) TERM *****
SUM = A(I,M)
DO 10 K = I+1, N
10      SUM = SUM - X(K)*A(I,K)
X(I) = SUM
20 CONTINUE
RETURN
END
```

The procedure we used is referred to as *Gaussian elimination* because, as we can see, after normalization and appropriate subtractions, the unknown  $x_1$  is eliminated from the second equation,  $x_1$  and  $x_2$  are eliminated from the third equation, and so on.

## Determinants

The determinant of a matrix may be used in solving systems of linear equations, but it is much more time-consuming than the methods we are covering for systems of order higher than two or three. The determinant of a matrix A, referred to as  $\det(A)$ , is defined as the sum of all signed simple products from A. The positive products are represented by the right arrows in the diagrams, and the negative product terms by the left arrows. For an  $n \times n$  matrix, these are the products of  $n$  entries from the matrix such that no two of the terms of the product are from the same row or same column. For example, the determinants of the following matrices of order 2 and 3 are as indicated:

$$\begin{aligned} \det \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} &= a_{11} a_{22} - a_{12} a_{21} \\ \det \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} &= a_{11} a_{22} a_{33} + a_{12} a_{23} a_{31} \\ &\quad + a_{13} a_{21} a_{32} - a_{13} a_{22} a_{31} \\ &\quad - a_{12} a_{21} a_{33} - a_{11} a_{23} a_{32} \end{aligned}$$

In cases where the determinant must be evaluated, note that if the matrix is put into (upper or lower) triangular form, the product of the elements on the major diagonal is the only nonzero entry in the sum of products that define the determinant. Thus, putting a matrix into triangular form is one good way

to evaluate the determinant. It only remains to calculate the product of the terms on the diagonal (ignoring the  $n+1$ st column of the augmented version of the matrix).

## FORTRAN 90 FEATURES

Because this chapter deals with sophisticated numeric techniques that may be sensitive to the range of values that can be stored on your machine, the number of significant digits possible to achieve, etc., the following new functions available in Fortran 90 to test system parameters, perform special manipulations, etc. are of interest in this context.

**CEILING (A)**—returns the smallest integer that is greater than or equal to the real argument A

**FLOOR (A)**—returns the largest integer that is less than or equal to the real argument A

**DIGITS (G)**—returns the number of significant digits on the system for the variable of type G—if G is real, it will return the number of significant digits on the system for reals; if G is integer, it will return the number of significant digits available for integers on the system

**EPSILON (X)**—returns a real value that, on the system, is almost negligible compared to 1 (argument X must be real)

**HUGE (G)**—returns the largest value on the system of the type (integer or real) specified by G

**MAXEXPONENT (X)**—returns the maximum exponent on the system

**MINEXPONENT (X)**—returns the minimum exponent on the system

**PRECISION (G)**—returns the decimal precision on the system for a value of type G (real or complex)

**RANGE (G)**—returns the decimal exponent range of the system for the type of value represented by G (integer or real); if G is integer, the value is INT (LOG10 (huge) ); if G is real, the value is INT ( MIN (LOG10 (huge), -LOG10 (tiny) ) )

**TINY (x)**—returns the value of the smallest positive real number that is representable on the system

Note that there are additional new matrix manipulation functions (such as ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCT, SUM, DOT\_PRODUCT, MATMUL, CSHIFT, EOSSHIFT, TRANSPOSE, MERGE, PACK, SPREAD, UNPACK, and RESHAPE) available in Fortran 90. There are also new bit manipulation functions available. These are covered in Appendix E.



## SUMMARY OF IMPORTANT CONCEPTS INTRODUCED IN THIS CHAPTER

In this chapter, we examined numeric methods for root finding (iteration, bisection, Newton-Raphson), least-squares fitting of data, numeric integration (rectangular quadrature, the trapezoidal method), differentiation, solving systems of linear simultaneous equations (Gaussian elimination), and finding determinants. We also examined methods of data improvement (enhancement, smoothing, reduction, and filtering).



## EXERCISES

**1.** Early in this chapter, in discussing the bisection method of finding roots of an equation, we spoke of finding two reasonable points between which the function changes sign. Given a function for which we need to find the roots, write a routine that will find two values for which the function changes sign, and then supply them to a BISECT function. Try an algorithm something like this. Pick a value, and begin moving away from the value in one direction. If you are going “uphill” and you already were at a positive value for the function, reverse direction. Using some step size, repeat in the “downhill” direction until the function changes sign.

**2.** Using the Newton-Raphson method, find the root(s) of:

$$\begin{aligned}x e^x - 1 \\x^3 - 2x + 1 \\e^x + 2^{-x} + 2 \cos(x) - 6\end{aligned}$$

- **3.** Assuming you have been given a data table of 100 values in a real array, write a subroutine which will “smooth” the values in the array by replacing each element (except the first and the last) by the average of the three elements of which it is the middle value. Try this two ways, first with a progressive “smooth” that takes newly smoothed values into account in determining new values, and then with a procedure that smooths only the original values in the array (you may need to use a second array temporarily).
- 4.** *Enhancing data.* Write a subroutine which will take a  $10 \times 10$  array of data brightness values and magnify it to be a  $40 \times 40$  array, by taking each data point and replicating it in a  $4 \times 4$  block of the new array.

- 5.** *Reducing a matrix of data.* Take a  $20 \times 20$  array and reduce it to a  $5 \times 5$  array by simply sampling every 4th row and column for your data values for the new array. Alter this routine to store, for every 4th row and 4th column position, the average value of the pixels in the  $3 \times 3$  array for which its position is the center value.
- **6.** *Removing noise.* Assume you are given a  $50 \times 50$  array of brightness data. Accept a threshold value, above which you deem a value to be a noise “spike.” Replace each spike value by the average of the values in a  $5 \times 5$  square surrounding it, but do not include the spike value in the average.
- 7.** Assume you are given a  $40 \times 40$  array of pixels. Find the largest and smallest brightness values in the array, and then use these to “stretch” all the values of the array to a range from 0 to 255, as explained in the text.
- 8.** *Convolution.* Assume you are given an  $N \times N$  pixel matrix, and an  $M \times M$  mask of weights to pass over the matrix to filter the data ( $N >> M$ ).  $M$  is probably 3, 5, or 7. Write a subroutine which will accept both arrays, and *filter* the data in the large array by passing the weighted mask over the array and replacing each value in the large array by the weighted average (determined by the  $M \times M$  mask) of its neighbors.
- 9.** *Enhancing Contrast.* Assume you are given an  $N \times N$  pixel array and a threshold value, such that all values below the threshold in the matrix are to be set to 0. Implement this.
- 10.** Do a little statistical research project to determine data for a scatter-plot of one variable with respect to another (for example, the heights and the weights of your acquaintances). Use a least-squares fit analysis to determine the best-fitting line to the data, and then use it to predict a new height given a weight (or vice versa). Plot the scattergram and include the best-fitting line on the plot.
- 11.** *Finding areas.* Using the three different integration methods discussed in Chapter 13 (rectangles, trapezoids, and Monte Carlo techniques), find the area under each of the following curves from  $x = 1$  to  $x = 2$ . Vary the number of subdivisions you use. Since these functions can be integrated by the methods of the calculus, compare your results with the analytic results.

$$f(x) = 1/x^2$$

$$f(x) = 4x^3 - 3x$$

$$f(x) = e^{2x}$$

$$f(x) = \sin(x) + \cos(x)$$

- 12.** Finding areas between curves. Using your favorite numeric integration technique, find the area, from  $x = 1$  to  $x = 2$ , between the two curves described by the functions:

$$f(x) = 2*x**2 - 4*x + 6$$

$$g(x) = -x**2 + 2*x + 1$$

- 13.** Using the limit definition of a derivative, and small values of  $a$ , approximate the *derivatives* of the following:

$$f(x) = \sqrt{x} \quad \text{at } x = 4.0$$

$$f(x) = x^4 - 3x^2 \quad \text{at } x = 2.0$$

Since these functions can have their derivatives determined by the methods of the calculus, compare your results to the actual analytic results.

- 14.** *Systems of Linear Equations.* Using simple Gaussian elimination, solve the following system of simultaneous linear equations. Compare your results with the actual solutions determinable by other means.

$$x_1 + 2x_2 + 3x_3 = 5$$

$$2x_1 + 5x_2 + 3x_3 = 3$$

$$x_1 + 8x_3 = 17$$

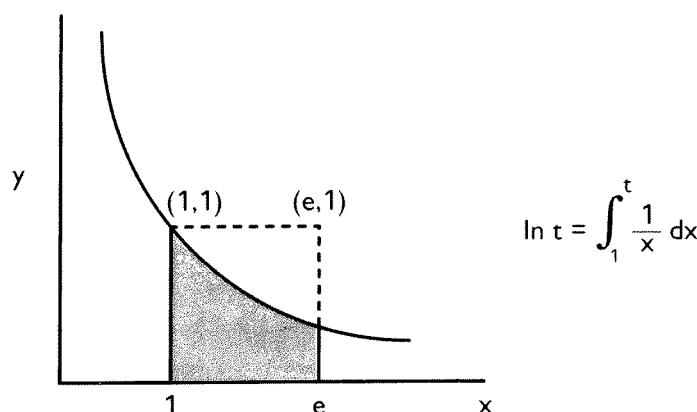
(Solution:  $x_1 = 1$ ,  $x_2 = -1$ ,  $x_3 = 2$ )

- 15.** In Chapter 9, we saw approximation formulas for square root and cube root. In general, the iterative formula for the  $n$ th root of a (positive) quantity  $A$  is:

$$X_{k+1} = 1/n ((n-1) X_k + A/(X_k^{n-1}))$$

Write a general-purpose function ROOT which accepts two arguments,  $N$  and  $X$ , and returns the  $N$ th root of  $X$ .

- 16.** Use numerical techniques for integration from this chapter to approximate the area representing the integral:



The area is that bounded above by the curve  $y = 1/x$ , below by the  $x$ -axis, on the left by the line  $x = 1$ , and on the right by the line  $x = t$ . The base of the natural logarithms,  $e$ , is the value of  $x$  ( $t$ ) for which this area is equal to 1. Use a value of  $e$  to as much precision as your machine will allow (you can get it using the EXP function,  $E = EXP(1.0)$ ); then use your technique to approximate the area under the curve from  $x=1$  to  $x=e$ , and see how close it comes to 1.0. Compare your results with those obtained by using a Monte Carlo approximation to the area (as discussed in the previous chapter).

- 17. The most efficient way to calculate  $x^n$ , for an integer  $n$ , is by reducing it to a recursive problem where the new quantity can be achieved by multiplying the current value by itself. Note that  $x^{12} = x^4$  times  $x^8$ , and  $x^{21} = x (x^4)(x^{16})$ , and so on. If  $n$  is odd, calculate  $x$  times  $x^{n-1}$ . The procedure for calculating the power is to set  $xx = x$  and  $prod = 1$ , and while  $n > 0$ , [ (if  $n \bmod 2$  is 1,  $prod = prod * xx$ );  $n=n/2$  (truncated),  $xx = xx * xx$  ]. Try it by writing a function for  $x^n$ .
- 18. In the book, *The Limits to Growth* (New American Library, 1972), the report of the Club of Rome on the future of the planet, one of the aspects they examined was the future capabilities of food production. To grow food requires *arable* (tillable) land, and the report's estimate is that there are only about 3.2 billion hectares of land potentially suitable for agriculture on the earth (a hectare is roughly 2.471 acres). This number will be further decreased over time through the use of power lines, paving, houses, and erosion, which will make it unsuitable for agriculture. The report indicates that 0.4 hectares per person is adequate to feed the world's population, but at U.S. standards, it would take 0.9 hectares per person! Current trends in population growth indicate that the amount of land needed  $t$  years after 1950,  $A(t)$ , can be represented by:

$$\frac{dA}{dt} = kA$$

If in 1950 the population needed  $1 \times 10^9$  hectares, and in 1980 it was estimated to need  $2 \times 10^9$  hectares, derive the formula for  $A(t)$ , and then write a FORTRAN program that will output how many hectares will be needed by any specified year after 1950. Can you also estimate in what year the need will reach the total amount of available land?

- 19. *The Limits to Growth* also presented a table (pp. 64–67) of the known global reserves (as of 1972) of various resources and the "Static Index" which estimates the number of years (after 1972) that resource would last at 1972's rate of consumption. However, the rate of consumption is increasing every year, so the depletion will not be linear, but exponential. The table also gives the estimated use growth rates (high, average, and low). Use this information to write a program that will estimate, for any input resource, amount in 1972,

static index, and specified growth rate, how much will be left (in terms of actual amount and expressed in terms of ratio of the 1972 reserves) by a specified year. Then add a subroutine to your program which will compute the year in which we will run out of a specified resource, given a particular growth rate.

Resource	Known Reserves	Static Index	Proj. High	Growth Av.	Rate Low
Aluminum	$1.17 \times 10^9$ tons	100	7.7	6.4	5.1
Coal	$5 \times 10^{12}$ tons	2300	5.3	4.1	3.0
Copper	$308 \times 10^6$ tons	36	5.8	4.6	3.4
Iron	$1 \times 10^{11}$ tons	240	2.3	1.8	1.3
Lead	$91 \times 10^6$ tons	26	2.4	2.0	1.7
Natural Gas	$1.14 \times 10^{15}$ cu. ft.	38	5.5	4.7	3.9
Petroleum	$455 \times 10^9$ barrels	31	4.9	3.9	2.9
Zinc	$123 \times 10^6$ tons	23	3.3	2.9	2.5

Even if these numbers were not accurate, your program could work on the same problem just by inputting newer values for the same parameters. However you look at it, it is frightening!

**20.** The radioactive element carbon 14 has a half-life of 5730 years. The amount of carbon at time  $t$  is then given as:

$$C(t) = C_0 e^{-kt}$$

Use this information to derive the value for  $k$ . Then write a program that will accept the percentage of carbon 14 found in an unearthed artifact (an Egyptian scroll, a Sumerian decorated box such as the Standard of Ur, or the like) compared to the ratio to be found today in living matter, and print out the estimated age of the artifact.

**21.** In Chapter 13, we studied the characteristics of the normal (Gaussian) distribution. If the average height of adult males in the United States is 5' 9"

and the standard deviation is 0.2, with heights fitting a normal distribution, write a program to determine what percent of the adult male population is 6' tall or taller. How about 6' 6" or taller?

Use the same sort of analysis, assuming a normal distribution of the weights of newborn babies with a mean of 7 pounds 12 ounces and a standard deviation of 1 1/4 pounds, to write a program to determine the probability that a newborn's weight falls in some specified range (say, 8 – 8 1/4 pounds).

**22.** Ecology tells us of various *allometric* relationships between a particular physical characteristic in an animal and another characteristic (usually body size). This is expressed:

$$Y = a X^b$$

or

$$\log(Y) = \log(a) + b \log(X)$$

For example, we are told that the relation between heart rate and body mass in mammals has a value of  $b = -0.23$ , and the relation between resting metabolic rate (RMR) of mammals and their body weight is expressed by  $b = 0.73$ . Pick a representative point for each scale—say, for the dog, a body mass of 10 kg. and a heart rate of roughly 200 beats per minute; and, for the squirrel, an RMR (oxygen consumption in  $\text{cm}^3/\text{hr}$ ) of 100 for a body mass of 0.1 kg.). Create a nicely labelled log/log plot for each allometric relationship, with body mass varying from 0 to 1000 kg., heart rate from 0 to 1000 beats/minute, and RMR from 0 to  $10^6 \text{ cm}^3/\text{hour}$ . This information is based on Robert E. Ricklefs, *Ecology* (New York: W. H. Freeman and Company, 1990).

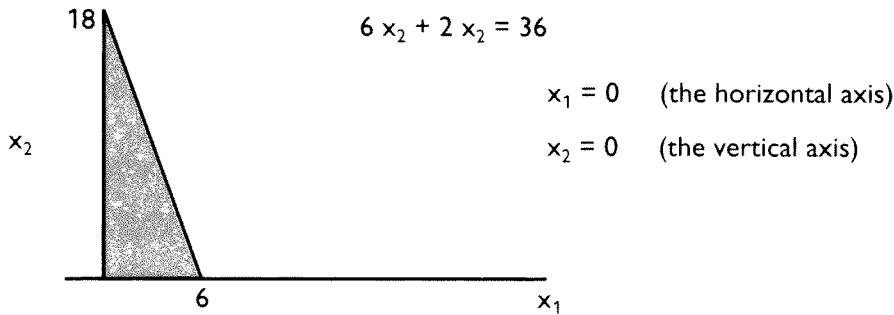
**23.** *Linear Programming.* In manufacturing and other processes, several factors (such as cost, time, labor, available material, etc.) often must be considered in combination in order to determine the most effective operation to be performed. These can basically be expressed in terms of linear inequalities. As a simple example, imagine a manufacturer of cardboard boxes who must create boxes in two sizes—one with an area of 6 square feet and another, smaller box with an area of 2 square feet. In addition, he only has 36 square feet of cardboard to use (obviously, this is a small business!). What are the possible combinations of boxes of the two different sizes that he can make? Obviously, this process is one in which no negative number of boxes can be made. Thus, if we let  $x_1$  represent the number of 6-square-foot boxes and  $x_2$  the number of 2-square-foot boxes, we are facing three concurrent linear inequalities:

$$6x_1 + 2x_2 \leq 36$$

$$x_1 \geq 0$$

$$x_2 \geq 0$$

In a simple situation such as this, there is a graphical solution to the problem. If we look at the area on a two-dimensional Cartesian graph that is bounded by the three lines:



it will represent the domain of possible combinations given the constraints. This is one way to solve such a problem.

Given your graphing skills developed in Chapter 12, write a program which will accept three similar inequalities defining a problem in two variables, and *graph* the domain of possible solutions. For example, your program should be able to accept the problem of a stereo manufacturer who must produce at least 500 VCRs and 800 TVs a week, and has workers who can make a VCR in 10 hours and a TV in 5 hours. He has 500 employees, each of whom works a 40-hour week, so the maximum time available is 20,000 work-hours. The relevant inequalities are:

$$\begin{aligned} x_1 &\geq 500 \\ x_2 &\geq 800 \\ 10x_1 + 5x_2 &\leq 20000 \end{aligned}$$

Your program should be able to draw a graph of the domain of possible solutions to this problem. Further, if the next problem is to maximize profit, and a \$50 profit is made on each VCR and a \$30 profit on each TV, add the feature that your program can evaluate this "profit function":

$$50x_1 + 30x_2$$

at each of the vertex points of the resulting triangle representing the domain of possibilities, and choose the one which has the maximum value for the profit function.

*The Simplex Method.* The problems of production and the interaction of criteria can be much more complex than what we were able to represent in our two-dimensional graphing method, which could handle two variables. George Dantzig, in 1947, developed the simplex method of linear programming for solving such complex problems. This method is just an extension of our earlier graphical approach to more dimensions than we can easily visualize. Though we will not take the space to detail this method here, it can be found in many texts on operations research or on finite mathematics, and we recommend such texts to you if you are interested in pursuing this application of your talents further. The matrix manipulation methods you have developed in this chapter will serve you well in such an endeavor. See, for example:

Larry J. Goldstein, David C. Lay, and David I. Schneider, *Modern Mathematics and its Applications* (Prentice-Hall).



## SUGGESTED READINGS

Acton, Forman S. *Numerical Methods That Work*. New York: Harper & Row, 1970.

Anton, Howard. *Elementary Linear Algebra*. 3rd ed. New York: John Wiley, 1981.

Atkinson, L.V., P.J. Hartley, and J.D. Hudson. *Numerical Methods with FORTRAN 77: A Practical Introduction*. Reading, Mass.: Addison-Wesley, 1989.

Cheney, Ward, and David Kincaid. *Numerical Mathematics and Computing*. 2nd ed. Monterey, Calif.: Brooks/Cole, 1985.

Edgar, Stacey L. *Advanced Problem Solving with FORTRAN 77*. Chicago: SRA, 1989.

Grandine, Thomas A. *The Numerical Methods Programming Projects Book*. Oxford: Oxford University Press, 1990.

Hultquist, Paul F. *Numerical Methods for Engineers and Computer Scientists*. Menlo Park, Calif.: Benjamin/Cummings, 1988.

Jensen, John R. *Introductory Digital Image Processing: A Remote Sensing Perspective*. Englewood Cliffs, N. J.: Prentice-Hall, 1986.

Kahaner, David, Cleve Moler, and Stephen Nash. *Numerical Methods and Software*. Englewood Cliffs, N. J.: Prentice-Hall, 1989.

Kuo, Shan S. *Computer Applications of Numerical Methods*. Reading, Mass.: Addison-Wesley, 1972.

Lillesand, Thomas M., and Ralph W. Kiefer. *Remote Sensing and Image Interpretation*. 2nd ed. New York: John Wiley, 1987.

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

Vetterling, William T., Saul A. Teukolsky, William H. Press, and Brian P. Flannery.  
*Numerical Recipes: Example Book [FORTRAN]*. Cambridge: Cambridge University Press, 1988.

# CHAPTER 15



## SOME CLOSING REMARKS

An adherence to high standards for structured FORTRAN programs will pay off in the efficiency and long life of the programs. Good style (as in literature) will make your work more readable (and also easier to maintain). Good documentation will help those who must use your programs as well as those who must maintain them. Software engineering considerations emerge in the writing of large programs by teams of programmers. The "Grand Challenges of Science and Engineering" for the foreseeable future (as outlined in a report to the President) will involve the use of FORTRAN-like programs on supercomputers.

The CRAY Y-MP/832 computer system is the top-of-the-line supercomputer of Cray Research, Inc. It contains eight central processors and 32 million 64-bit words of memory.

*"We must all hang together or, most assuredly, we shall all hang separately."*

- Benjamin Franklin, Remark to John Hancock at the Signing of the Declaration of Independence, 4 July 1776



## STANDARDS FOR FORTRAN PROGRAMS

The notion of standards (that is, criteria of excellence) for FORTRAN programs has changed considerably over the years. Emphasis used to be placed primarily on program efficiency. Now the emphasis is much more on programmer efficiency (computer time costs less and is much faster than it used to be, and programmer time, you may be glad to hear, has become quite high-priced). Thus, the following two sections really sum up much of what is now considered part of the description of an excellent program—one written using structured methods, and which has good style. But the idea of efficiency should not be left out of consideration altogether. We have stressed throughout the text more efficient methods of doing things. We conclude this chapter with a section on program optimization, which is important for large-scale, time-consuming programs such as those run on supercomputers (discussed in the penultimate section) and for "real-time" applications.

Yet there is more to the notion of standards for a good FORTRAN program than structure, style, and efficiency. Good use should be made of existing tools, such as system functions and libraries of subroutines—for example, those provided by IMSL and NAG (references for these sources are given at the end of the chapter). The programmer should also develop a set of tools, subroutines and functions that will have a continuing life. This has been emphasized throughout the text, and is further discussed in the section of this chapter on software engineering.

A "standard" for FORTRAN should involve an adherence to the FORTRAN ANSI Standard itself. Working within the Standard will pay off in greater portability for your program, from one machine to another. If you must make use of a system-dependent feature, flag this clearly in your program, and even provide notes as to how it could be changed to run on a machine that does not have this non-standard feature. If the Standard does not provide WHILE and REPEAT/UNTIL structures (as FORTRAN 77 does not, and Fortran 90 does not include a Repeat Until structure), then we have seen that there are ways to simulate these structures. These simulations can create the control structures you want and remain within the Standard.

Similarly, for program compatibility, do not assume that the computer "clears" (that is, zeros) memory between programs. Be sure to initialize all your variables before they are used. Do not "mix modes" unless necessary, since this requires the compiler to perform type conversions. Thus, if A is a real variable, write  $A = A^*2.0$  rather than  $A = A^*2$  so that no conversion of the integer constant 2 to real is required.

All declaration statements (type, dimension, common, equivalence) must appear at the beginning of your program. Any statement functions must come after these and before the first executable statement in the program. DATA statements should come after the declaration statements, and should not be mixed in with the executable statements; if they are, it makes the program more difficult to read, and gives the erroneous impression that DATA statements may be executed as part of the program logic flow.



## STRUCTURED PROGRAM DESIGN

*Structured programming* is an approach, a “philosophy,” to writing programs that will be correct, easy to use, easy to verify, easy to modify, and easy to read (understand). In 1976, E. W. Dijkstra wrote a book called *A Discipline of Programming* describing this approach; we may well view structured programming as *disciplined* programming. If programmers are, to a large extent, using the same programming language constructs in the same ways, then they will find that it is relatively simple to read, understand, use, maintain, and modify one another’s programs—this is one of the major goals of structured programming.

It has become evident in recent years that many programming problems are of too large a scope to be solved by one person; they must be worked on by a “team,” working on separate components of a larger structure that can at some point be brought together in a coherent manner. Imagine, as an analogy, a team of workers building a house, where basically each group of the workers is given a room to design and build, plus some special-purpose workers who design plumbing, lighting, or heating which may be used by the several room-builders.

The house-building analogy should make it clear that, although the room-builders can largely work independently of one another, there must have been an overall design strategy as to how the rooms will fit together, and which precludes the possibility of redundancy. Each room-builder must have some concern about the *interface* of this room with the other rooms—doors must be in the same place on a common wall between rooms and so on. The plumbers and electricians service several of the room-builders, and the latter must know how to access their services. The plumbing and electrical work are analogous to functions that may be referenced by the various room-builders. The workers should use similar terminology and techniques, so that they can communicate with one another and resolve conflicts or misunderstandings if they arise.

Structured programming proceeds from a *top-down* design analysis of the problem into smaller problems. The overall design can be looked at and checked out *on the assumption that* the subproblems will be solved (on the house analogy, that the individual rooms will be completed). The actual process of completing the subtasks is their *stepwise refinement*—they are analyzed down to the level where “atomic” tools exist for their solution, and then the solution

can be built up, also step by step, using these tools.

In a pure "block-structured" language, with all of the necessary design tools, it can be proved that any problem that can be solved on the computer can be solved by using just three kinds of control structures: *sequence* (steps will be executed in order), *selection* (like the various IF structures), and *repetition* structures (like the DO). Notice that no branch statements (such as GO TOs) are mentioned here. But you must also realize that these higher language structures can only be built up by using branch statements at the lower (machine-implementation) level, just as we had to use IFs and GO TOs to simulate the "while" and "repeat until" structures, since they are not standardly available in FORTRAN 77. In a true *block-structured* program, there should also be only one entry and one exit to each block of code that does a job. This is more difficult to adhere to in a language like FORTRAN, which does not have all of the repetition structures built in. To compare the single-entry/single-exit form to one which is not handled this way, imagine programming the problem of continuing to add up the fractions in the series:

$$1 + 1/2 + 1/3 + 1/4 + \dots$$

until the sum exceeded 4.0, or until the first 100 terms had been added, whichever comes first (this is analogous to a problem we looked at in the first section of Chapter 4). One way to do this would be using a DO loop that allows for a premature exit if the sum exceeds 4.0, but continues adding terms otherwise:

```
SUM = 0.
DO 20 N = 1, 100
    SUM = SUM + 1.0/N
    IF (SUM. GT. 4.0) GO TO 25
20 CONTINUE
25 PRINT*, 'THE SUM IS ', SUM
```

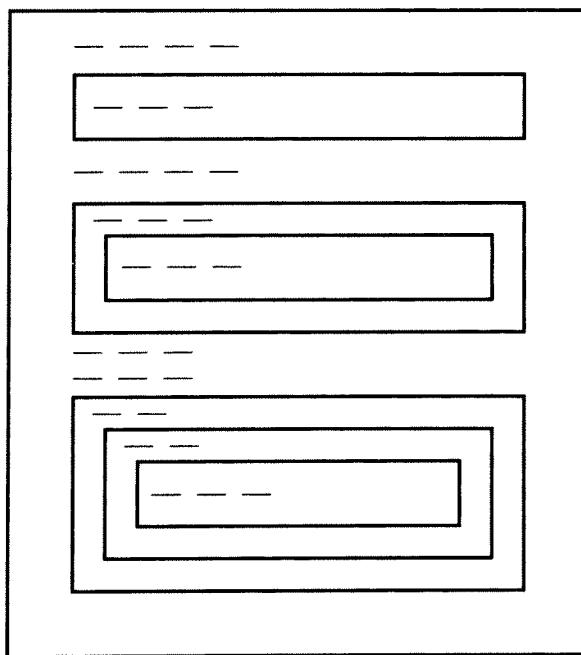
This loop has *two* possible exits, one the normal exit from the loop after 100 terms have been added, and the other the premature exit if the sum exceeds 4.0. (Draw a diagrammatic picture, such as a flowchart, of this.) If we adopt the premise of the ultimate desirability of a *block structure*, in which our loop (block) should have only one entry and one exit, we will have to rewrite the code to do this. What we want is a repetition structure that will repeat as long as the counter N is less than or equal to 100 and the sum is less than or equal to 4.0. We can implement this using a simulated WHILE structure and a logical connective .AND. for the two conditions that must be satisfied for the loop to repeat:

```
SUM = 0.
N = 1
***** WHILE (N.LE.100 .AND. SUM.LE.4.0) DO *****
15 IF (N.LE.100 .AND. SUM.LE.4.0) THEN
```

```
SUM = SUM + 1.0/N  
N = N + 1  
GO TO 15  
ENDIF  
PRINT*, 'THE SUM IS ', SUM
```

Thus, we *can* implement the one-entry/one-exit block form if we so desire. Note that if we wanted to know *which* test was the one that got us out of the loop, both solutions would have to be modified. Think about how you would modify both forms to print out if the sum of 100 terms, or that you just summed up to 4.0 and stopped. (A simple way to do this might be to print the value of N as well as that of SUM [or of N-1 in the second case], to indicate how many terms had been added up. However, what if in a more complex problem it was important to know *which* test caused the exit, and no such simple way to determine this?)

A *block-structured* program can be viewed as one made up of block structures (Block IFs or loop structures), each of which has one way in and one way out. Interspersed among the blocks there may be simple sequential commands, such as assignment statements or I/O. Each block may contain sequential statements, and it may contain other blocks, but each "nested" block must be completely contained within the outer block (as you recall, we would not allow the *ranges* of our Block IFs and/or our DO loops to cross). Thus, the overall structure of the program may be envisioned as sequential statements and blocks, in a structure that might look something like the following:



A structured program will employ structured control tools (such as Block IFs and repetition structures) and sequential statements in a systematic, "disciplined" way, such that the logic flow of the program is clear and easy to follow. It will also employ individual data items (constants and variables), and structured data types. The basic structured data type in FORTRAN is the array. Out of this we can build other structured data types, such as linked lists, stacks, and queues, as we have seen. A good problem-solving approach will choose both the algorithms (and the control structures that implement them) and the data structures wisely.

Clearly, a structured program will make use of *modularity* —that is, it will use top-down design to divide the problem into simpler parts, many of which may be implemented in subroutines or functions. Modularized programs are easier to design and test. Modules may be tested independently, before they are incorporated into the program, and the overall structure can be tested early for its integrity by using "stubs," as we have described earlier. Modularity also makes the program easier to *modify*, should the need arise, since the change(s) may only need to be made to one or two subprograms. This gives the program an element of *generality* as well.

Program modules (subproblems implemented in subprograms) should be made as independent of one another as possible, so that they are easy to modify without having to change other whole sections of the program as well. Furthermore, the more independent they are, the more readily they can be fitted together to form the whole, even if they are designed by different programmers. A person writing the main program or another subprogram needs only to know *what* the other subprograms do, not *how* they do it—this is called *information hiding*. This feature is particularly important in the design of programs where security is an issue, such as military programs.

Knowing FORTRAN well allows you to use it well!



## WRITING WITH STYLE

Good programming *style* is very closely connected with the topics of the two earlier sections—FORTRAN standards and structured programming. A structured program has a good program style. But many other things go into making a good style, and we will try to highlight some of them here. You will find that you will develop a good style of your own, and you will decide that certain techniques are more important to emphasize than others. Your program style will not be quite as free as good writing style in a natural language like English, since the programming language and its structures are much more limited; but you will find a fair amount of flexibility even so, within which you can develop your own style.

Good style contributes to a clear, readable, understandable program. Using structured programming constructs and meaningful variable names and

subprogram names (so that the entity a variable stands for or the purpose of a subprogram is made clear by the name) enhance clarity. Well-thought-out comments contribute a great deal to program clarity. If you use meaningful names, you should not have to have a lot of comments. Make your comments clarifying, not redundant. Clearly explain any algorithm in use, the purpose of the program, and any special features you have employed in your comments. You may want your comments to include a "data dictionary" at the beginning of the program, which clearly lists (perhaps in alphabetical order) all of the important variables in the program and what they do.

A program should be well-*documented*, both internally and externally. Good comments and meaningful variable and subprogram names are an essential part of good internal documentation, as is readable, clear code.

Your program should be visually attractive and composed of "grammatical" parts. The use of structured components such as Block IFs and DO loops will contribute to the correct syntactical form. You should use blank lines between subprograms, so that they are easy to find. If possible, arrange the subprograms in the logical order in which the reader of the program would expect to find them (that is, in the order in which they are called). The FORTRAN compiler does not need them in any special order, but it will make things easier for a human reading the program.

To make your program easier to read, use indentation (since FORTRAN ignores blanks except in literal strings) to make sections of code readily identifiable. Throughout the text we have adopted the convention of indenting the body of an IF block or a DO loop three spaces, and having the beginning and end statements at the left margin for that structure (other control statements, such as ELSEIFs, in such a block also appear at the left margin for that block). This makes the sections belonging to a particular control structure clearly identifiable. For example, we would write two nested IF blocks as follows:

```

IF ( MOD(NYEAR, 4) .EQ. 0 ) THEN
    IF ( MONTH.EQ.2 .OR. MONTH.EQ.8 ) THEN
        NTOT = NTOT + 3
    ELSEIF ( MONTH.EQ.6 ) THEN
        NTOT = NTOT + 5
    ELSEIF . . .
        .
    ELSE
        NTOT = NTOT + 6
    ENDIF
ENDIF

```

Notice that this makes it very easy to see where each control structure begins and ends, and which commands are under the scope of certain control phrases, such as the ELSEIF.

Only use statement numbers on statements that absolutely need them, such as the terminal statement of a DO loop, or a FORMAT statement, or the first statement of a simulated *while* or *repeat/until* structure. This will make such statements stand out nicely in the program and easy to find. To make labelled statements even easier to find, number them in increasing order throughout the program. It is also advisable to make FORMAT statements appear different from other labelled statements; for example, you might label terminal statements of DOs and initial statements of simulated *whiles* and *repeats* with numbers that are multiples of 10, such as 20, 50, 100, and so on, and label FORMATS with numbers that do not end in zero, and preferably repeat, such as 22, 333, and so on. Be consistent in your placement of FORMAT statements in the program, so that they can be readily found. Put them all together at the end of the program, or at the beginning of the program, or (our personal preference) immediately after the first reference to them.

You can decrease even further the number of statement labels needed by using the method of describing the format as a character string directly in the I/O statement, instead of using a statement number and a FORMAT. For example, replace

```
PRINT 5, A, B, D
5   FORMAT (6X, 3F10.2)
```

by

```
PRINT '(6X, 3F10.2)', A, B, D
```

Further, if the DO/END DO structure discussed in Chapter 4 is available on your system (it will be standardly available with Fortran 90 compilers), you can eliminate using a statement number for the terminal statement of a DO.

Many programming stylists recommend that you *only* end a DO loop with a CONTINUE statement, and this certainly improves the coherence of the statements in the body of the loop. We have generally adopted this convention throughout the book, except in the case of one-statement ranges of a DO.

The following are some brief recommendations regarding style:

1. Use a PARAMETER statement to give symbolic, meaningful names to significant constants in your program. If a symbolic constant refers to the size of a group of items to be processed, and controls array sizes and loops throughout the program, this feature will make it very easy to change, if necessary.
2. Make use of the optional PROGRAM statement to give your program a name, and to identify its starting place clearly, in the same way that your subprograms have identifiers.

3. Use the STOP statement to indicate where the logic of your program terminates, and the END statement to mark the physical end of the program or subprogram. Similarly, use a RETURN statement in your subprogram to indicate the point at which it transfers back to the calling program unit; do not let the END statement do the work of the RETURN by default.
4. Use extra parentheses in arithmetic and logical test and assignment statements, to make the order in which operations are to be performed very clear.
5. When you pass an argument list to a subroutine, first pass those arguments that are going "in" to the subprogram, then those that are coming "out" (that is, those which will be defined by the subprogram), and finally those that are "inout" (that is, those that are shared in both directions, passed to the subprogram and redefined within it). Even though this will not provide any special control of these arguments in FORTRAN 77, it is a bookkeeping device that will help you to remember how the arguments are used. Furthermore, it is a good way to think of them, grouped together, which will make the transition to Fortran 90 (where you will be able to explicitly declare them IN, OUT, or INOUT) an easy one to make.
6. Use type statements to declare array bounds, rather than the DIMENSION statement (which may be removed by the Fortran revision around the turn of the century).
7. Always use subscripts when identifying an array element in an assignment statement or the like, even if the compiler would assume subscripts (such as 1s) if you left them out. This makes your intention much clearer, and it is not subject to inconsistencies among different compilers.
8. When you are doing I/O on an entire one-dimensional array, or on a two-dimensional array in column by column order, it is more efficient (as well as easier for you) simply to use the array name with no subscripts, rather than an implied list. Thus, for example, for the array KAT, use:

```
INTEGER KAT(100)
READ 66, KAT
```

rather than

```
READ 66, (KAT(I), I = 1, 100)
```

9. Be consistent in the number of decimal places you specify for real values. If they are simply real versions of integer values, to avoid mixed-mode conversion, use only a decimal point (2.) or, preferably (for readability), a decimal point and one zero (2.0). For reals with more significant

fractional digits, be consistent for different reals in your program, and stay within the significance limits that can be expressed by your machine (for example, 3.14159, 22.9876).

10. Make it clear when you use parallel arrays to simulate a record-like structure (discussed in Chapter 13), that these arrays *are* related, and should retain compatible dimensions, be changed, be sorted, and so on, in parallel. You can do this by grouping them together in the dimensioning statements, using a PARAMETER symbolic constant (or constants) to define the array dimension(s), and by using clear comments to convey their relation. For example,

```

PARAMETER (NEMPS = 77)
***** NEMPS INDICATES NUMBER OF EMPLOYEES REPRESENTED *****
      CHARACTER NAME(NEMPS)*12, TOWN(NEMPS)*10, STREET(NEMPS)*15
      INTEGER SSNUM(NEMPS), CHILD(NEMPS), SALARY(NEMPS), AGE(NEMPS)
***** EMPLOYEE RECORD CONTAINS THE FOLLOWING DATA *****
***** NAME
***** SSNUM
***** STREET
***** TOWN
***** AGE
***** SALARY
***** CHILD
***** INPUT EMPLOYEE RECORD *****
DO 50 I = 1, NEMPS
    READ 88, NAME(I), SSNUM(I), STREET(I), TOWN(I), AGE(I),
&     SALARY(I), CHILD(I)

```

etc.

## Input/Output Style Hints

1. “Echo-print” your input data, as a check that it has been entered and processed correctly. Remember that if you do not include an output statement for the data that has been entered, you will never see it as part of the program results.
2. *Validate* data that is entered to the program. Make sure that it lies in an appropriate value range before you let your program go on to process it, or you may have the program “blow up” over out-of-range data. Make your program as “robust” as possible, so that it will not abruptly terminate if the naive user enters the wrong type of data. Ways to avoid this are discussed in a later section. If the data entered is of the wrong type, or out-of-range, output a message to the screen telling the user so, and re-prompting for a correct input.

3. *Label* all of your output results very clearly, so the casual reader of the output can understand them. Include a brief description of the output and its connection to the problem that was solved. Sometimes pictorial output is the most revealing and easiest to interpret (see Chapter 12).
4. Output from one program that is to be used later as input to another program should be formatted as simply as possible. If there is no need for a human to scan the output, it is even better to write and read it in *unformatted* form (see Chapter 11). This will save considerable I/O time.
5. If you are writing output to magnetic tape, which blocks records and inserts an *interblock gap* between each block of records to allow for the backup when the tape is started in motion, make the records as long as possible, to minimize space wasted in interrecord gaps.
6. CLOSE all your external files before terminating your program.

## DOCUMENTATION

A good program should have a long life. To make this life a fruitful one, other people should be able to use the program easily, understand how it works, and be able to modify it if the occasion should arise. This means that the program should have good documentation, so that those who use it and work to understand and maintain it will find their jobs simple. Documentation should be of two kinds—internal and external.

### Internal Documentation

Internal documentation is that contained in the program itself. A program written with good style, using meaningful variable names, well-structured code, and enlightening comments has good internal documentation. We have already discussed many of the components of good style in the previous section, and these will make the program easy to read. Good comments are essential to internal documentation. They should indicate the problem being addressed, the algorithm(s) used, and any special implementation features the program has.

If any system-dependent features are used (though it is best to avoid them if possible), their nature should be clearly explained in comment statements, and ways to substitute other methods for them on different systems should be suggested. Each subprogram should contain a clear description of its purpose. All important variables should be described in the beginning of a program unit. The date of composition of the program should be included, as well as the dates

of any modifications.

If debugging output statements have been left in the program, their purpose should be indicated and the way to activate them should be described. The original authors' names should be given, and (if possible) addresses at which they can be reached. All error-handling messages should be clearly marked, and the source of the error explained, in comments.

Some of the internal documentation may be in the form of carefully explained instructions given to the user. There is no need to repeat this information in comments, since it is there in the code for the person scanning the program to read.

Remember that the person who is reading the internal documentation is a FORTRAN programmer, much like you. You thus do not have to oversimplify explanations; you may assume a certain common base of knowledge about FORTRAN programs, and your comments should build upon this.

## External Documentation

External documentation may serve many purposes. It may be written to help the naive user run the program, or to "sell" the program as an integral part of a package delivered to a group that contracted for it, or it may be additional detail (besides the internal documentation) for the person who must maintain and modify your program. Each of these pieces of external documentation is of a different nature.

A *User's Guide* should be as simple and straightforward as possible, so as not to overwhelm the user who is not very "computer-literate." It should state clearly the purpose of the program, so it is easy for the user to determine whether your program is appropriate for his or her needs. Say *what* the program does, and *why*, and you might optionally include a short technical section on *how* it does it, for the occasional technically competent user. You may want to include some technical reference works in a sort of bibliography.

The user will input data and receive results. You should have made the input procedure as simple as possible (discussed further in the next section on software engineering), unless it must be tailored to already existing data your sponsor has. There should be clear instructions output by your program that direct the user how to enter the information the program needs. It is useful in the program guide to provide a sort of *template* for such input data, and an actual example of how it should look. Your program should be robust enough to recover from incorrect entries by the user, and the Guide should explain how the program will react in such cases and what the user will then be expected to do. If an error message may be prompted by a number of possible conditions, explain to the user what these might be.

The user may also have to interpret the output of the program, rather than just hand it over to someone else. Of course, the output should be well-labelled

and clearly described. In addition, the User's Guide should give a template of the form of the output, and an actual example of program output (which might be annotated for the user's information).

If the input is to be from tape or disk, be sure to give the user adequate information as to what type of tape (7-or 9-track, for example) is expected, or what is required in the case of an input disk file. If the file is sequential and requires an end-of-file, be sure to describe how that is to be accomplished. Most control information will be entered by the user from a terminal keyboard. Be sure you describe how to correct an error on the keyboard before sending a command to the program. The user will greatly appreciate this.

A *report* on the program to a sponsor should be brief and to the point. It should explain the problem the program solves, and how it does it. Example output might be shown, with additional explanatory comment. For detail on actually running the program, refer the reader to the User's Guide. Include a reference list of technical sources you used. Indicate any ways in which you think the program might profitably be modified or expanded. If possible, give representative running times for the program.

A *maintenance guide* should include any additional information (besides that contained in internal documentation in the program itself) that you feel might be helpful to the person who "inherits" your program. A flow chart, or Nassi-Schneiderman diagrams, or whatever development tool you used, would be helpful. Describe any major difficulties you had with the program, and how you solved them. Give suggestions for changes to the program you think would be beneficial, or ways in which it could be expanded (as in the report). Refer the programmer to the user's guide for additional information. Try to put yourself in the position of the person who is given this program, and must maintain and update it. What would *you* want to know?



## SOFTWARE ENGINEERING

Engineering problem solvers combine knowledge of physical principles and mathematical techniques with a dash of creative genius to unravel a wide variety of real-world puzzles. In the area of software development, much can be learned from the engineer's approach to problems, and this view has given rise to an appellation of *software engineering* to "the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works on real machines." (Fritz Bauer, in "Software Engineering," *Information Processing 71* (Amsterdam: North Holland, 1972), 530)

A "software crisis" was seen to arise in the 1970s, when the size of programs needed to solve problems seemed to have gotten beyond the control of program designers. A new approach to handling large-scale problems was needed, one

which could assure quality and accuracy for programs of 100,000 to a million lines of code and more. Grady Booch, in his book *Software Engineering and Ada* (Benjamin Cummings, 1987), indicated a need for such programs to be modifiable, efficient, reliable, understandable, timely, inexpensive, reusable, responsive, and transportable. Not much to ask of a million-line program!

We have already addressed questions of modifiability, reliability, understandability, and transportability in the earlier sections of this chapter. A structured approach to problem solving, a program written with good style and which conforms to high standards and to the ANSI Standard has a much better record on these criteria than earlier, more "seat-of-the-pants," programs. We will address questions of timeliness and efficiency in the upcoming section on optimization, since these considerations are also a part of software engineering. Booch also says that these programs should be inexpensive, responsive, and reusable. We shall discuss these specific characteristics and how software engineering attempts to achieve them in the rest of this section.

Engineers have a wide variety of reusable *tools* at their disposal, which range from calculating tools like slide rules and computers to the mathematical tools of the calculus, statistical analysis, and the like. Once an engineer has mastered one of these tools, it becomes a permanent part of his or her repertoire. Tools are developed and honed to perfection, and then used over and over again. This analogy is important in software engineering; the programmer should develop tools which are thoroughly tested, and then reuse them whenever it is appropriate. We have stressed throughout this text the development of various FORTRAN problem-solving tools, from array handling (summing, averaging, sorting, searching, and so on) to simulation tools and numerical analysis tools.

Many of these tools (approaches) have been written as subroutines or functions, and in writing these we have attempted to obtain as much generality as possible. A subroutine written to average real arrays is not written just for arrays of length 100, but for arrays of any length; a subroutine to integrate the area under a curve is written so that it will work on any well-behaved function of one variable, between any specified limits; and so on. Once these tools are developed, we do not throw them away. They are maintained, and dusted off for use every time a problem that could use them (or even an analogous problem which they can be modified to fit) arises. The miracle of modern file saving and copying techniques allows us to maintain an original tool (subprogram) for a job, and also copy it into any new application, or create a copy we can modify for a new problem which we see as analogous to, or a variation on, the old one.

The reusability of such tools takes care of two of the items on Booch's list—reusable and inexpensive (since expensive programmer time does not have to be spent redeveloping a tool already in our collection).

The characteristic of *responsiveness* is a little more difficult to capture, but it seems to lie in the "user-friendliness" (a very overworked term) of the program. The program should be written with the user in mind (and, if the precise

user is known, you should try to fit the program to that user). A program tool written for continued use should be flexible enough to allow for easy modification, and robust enough not to be brought to its knees by the first user error. Thus, the program or program unit (subprogram) should incorporate ways of rebounding from user entry errors. First of all, the form in which the input is to be entered should be made as simple as possible, to minimize the chance of the user making an error. Then, tests will still have to be incorporated to be sure that the entry is of the correct type and does not represent a value out of the appropriate range.

## Rebounding from User Entry Errors

The simplest error to handle is an out-of-range value, since it can be detected by a simple test, and the user can be reprompted to enter a legal range value. For example,

```

40  CONTINUE
      READ 44, TEMP
44    FORMAT(F10.2)
      IF (TEMP.LT.-100.0 .OR. TEMP.GT.400.0) THEN
          PRINT 55
55    FORMAT(2X, 'YOUR VALUE IS OUT OF RANGE! //2X,'PLEASE
&      ENTER A TEMPERATURE BETWEEN -100.0 AND 400.0. THANKS.')
          GO TO 40
      ENDIF

```

A more difficult problem is when a value of the wrong *type* is entered. Normally, if the program tries to read, say, a real value or a character string with an integer format, an error condition will occur, and the program will terminate abruptly. There are two approaches to avoid such program termination. One is to use the ERR= clause in your READ statement, and use it to branch to a part of the program that prints a message to the user that the data entered does not fit the template described, and reprompt the user for a new entry.

Another approach is to read the entry line with a character format (A format), since *all* strings are legal character strings. You can then test the character string to see if it represents the proper type of value that was to be entered. If it does fit the proper form (say, it is all digits in the field that was to be read, for an integer value), then you can use core-to-core reformatting (Chapter 11) to put it into proper form. For example, suppose an integer is to be read into variable NUMB:

```

CHARACTER LOOK*10
LOGICAL BAD

```

```

30  CONTINUE
    READ 33, LOOK
33  FORMAT(A10)
    BAD = .FALSE.
    I = 1
40  IF (I.LE.10 .AND. (.NOT. BAD)) THEN
        IF (LOOK(I:I).NE.' ' .AND.(LOOK(I:I).LT.'0'.OR.
&      LOOK(I:I) .GT.'9')) THEN
            BAD = .TRUE.
        ENDIF
        I = I + 1
        GO TO 40
    ENDIF
    IF (BAD) THEN
        PRINT*, 'WRONG DATA ENTERED. TRY AGAIN'
        GO TO 30
    ENDIF
    READ (LOOK, 44) NUMB
44  FORMAT(I10)

```

Similar tests can be constructed to test for the legality of any input value entered.

From a user's point of view, the program is a "black box" (another concept borrowed from engineering). The user is only concerned with the relationship between the *input* to the "black box" (the program) and the *output* it produces. The user has no *need* (and also, perhaps, in programs involving security considerations, no *right*) to know *how* the program does what it does. It is this aspect of "information hiding" we have discussed earlier that dictates that all communication between a main program and its subprogram should be through argument lists, and all communication between the user and the program should be simply in terms of input and output.

## Program Testing and Reliability

A program or program unit must be thoroughly tested if it is to become a software *tool*. This testing should begin with a "desk-check" of the program logic, before even putting it on the machine. In this, the programmer "plays computer" and runs a set of data through the program, doing everything the program says (not what you *intended* it to do). If this test works, the program should then be entered, and compiler errors such as typos corrected. Get a symbolic reference map and check all of the variables on the list to be sure that

they are all ones you intended to have (no typos in variable names that the computer merely takes to be different variables), and that all of them are properly initialized before they are used. When the program compiles and has passed your other tests, it should be subjected to actual data to test its correctness. This should begin with simple data for which you know the expected result (say, the values you ran in the hand-check), and then expanded to include randomly chosen data (see Chapter 13) both *in* the legal range and *outside* the range of expected values to be entered. This way you can check thoroughly to see that the program behaves properly, no matter what value is entered.

## Structured Walkthroughs

*Team efforts* to develop large-scale programs have given rise to what is known as the *structured walkthrough*. This is a group session in which the program design specifications, or the details of a particular segment of code, are analyzed for flaws and weaknesses. The point of a code review may be to track down a particularly evasive bug, or it may be just to obtain overall suggestions for improvement. The developer of the code (or of the program design) must make a presentation of the project so far. In a presentation of program design, everyone in the group looks for ways to improve it, make it more efficient, or include aspects that have been overlooked.

In a walkthrough for program code, those in the group are attentive to whether the code does what it is supposed to do, whether there are more efficient ways of accomplishing the goal, and whether there are any errors or oversights in the program. You are probably familiar with the phenomenon that it is much easier for someone else to read over your program, or a paper you have written, and find the errors, than it is for you to do so. This is because, in a sense, you are *too* familiar with the work, and know how it *should* be, so that you may overlook errors, typos, and so on, because you know what should be there and assume (or mentally fill in) that it *is* there.

Walkthroughs may also be conducted to *test* the program. A stranger to your program may be able to think up more diabolical tests for your code, and be more willing to do so, than you are. Again, an advantage is gained by having the perspective of someone who is not *too close* to the program.

Another advantage of walkthroughs is that several people on the team become familiar with various parts of the project. This may help them better integrate their own component with yours and with the overall program. It also makes it less likely that one person is *indispensable* to the project. If one person leaves, there are several who have participated in walkthroughs on that program who can fairly easily pick up where he or she left off.

## The Software Life Cycle

The percentage of cost of a computer project used to be largely in the hardware, with the software representing a small fraction of the cost. This has undergone a fairly rapid reversal in trend, so that now the cost of a computer solution to a problem lies largely in the software, with a very small fraction of the overall cost attributable to hardware. The software costs are very complex, and more involved than you might think.

The “software life cycle” is said to consist of the following stages: requirements specification (make sure they are *clear*), problem analysis, program and algorithm design, implementation (coding), testing, and maintenance. It may surprise you to hear that it is estimated that from 2/3 up to about 4/5 of the total time spent in this “life cycle” is in *maintenance*—keeping the program running, getting rid of previously undiscovered bugs, and making modifications.

The programs you have worked on in this book have been small-scale programs, ones that are fairly easily developed and tested (though you might not have thought so at the time). You may still write many short programs of similar nature during your career, to quickly solve a problem that comes up. However, at some time you may become involved in a large-scale project, one perhaps even for a supercomputer, in which these software engineering considerations will be very important.



## SUPERCOMPUTING AND FORTRAN

The “wave of the future” in scientific computing seems to be the *supercomputer*—the class of a select few of the fastest, most powerful computers in operation. Certain problems seemed for a time to be practically insoluble, because too much computation time and/or too much memory space was required to handle them. The supercomputer represents technical advances that are bringing more and more of these problems into the realm of possible solution every day.

Supercomputers are multimillion-dollar products, and only a small number are sold every year, to the government and a few large corporations, for solving *big* problems. Some of the kinds of applications for supercomputers include the following: nuclear research (such as that conducted at Lawrence Livermore Labs), quantum chemistry, rocket design, automobile and aircraft research and design (materials strength, safety, fuel efficiency, crash simulations, etc.), molecular dynamics (with applications in drug research), circuit design (computers designing computers), weather prediction, seismology studies to detect sources of fossil fuels, earthquake studies, oceanography, and computer animation for movies. Supercomputers are being used to study and make predictions regarding crucial concerns such as the greenhouse effect, depletion of the ozone layer, and “nuclear winter,” as well as in research for cures

for killer diseases such as cancer and AIDS.

Supercomputers are also used to implement some computer programs that are pushing the limits of *artificial intelligence* (usually defined as a computer performing an activity that would be considered intelligent if it were done by a human being) including areas such as computer chess. One such program is Cray Blitz, which runs on a Cray X-MP supercomputer.

There are serious challengers in programs that run on special-purpose, specially designed "chess machines," such as Hitech, developed at Carnegie-Mellon, which uses one VLSI chip for each square on the board, and had a 2407 ranking (in the International Masters range), and Deep Thought, also developed at Carnegie-Mellon (by students there), which recently won the \$10,000 Fredkin Intermediate Prize for being the first chess computer to achieve a ranking of 2500 or better in 25 consecutive games against human opponents. Its ranking places it among the top 30 players in the United States. Deep Thought uses two processors, each of which can evaluate 450,000 positions per second; the two working together can evaluate 700,000 positions per second. It is five times as fast as Hitech (Hans Berliner, "Chess Report," *AI Magazine*, 10, 2 (Summer 1989), 89-90).

Supercomputers have larger memories than regular mainframes (say, from 250 million to a billion words of storage), and they are much faster. A computer's speed is measured in FLOPs (that is, floating-point operations per second), and since all can perform over a million floating-point operations per second, the normal measure is in megaflops. A supercomputer can perform around 500-600 megaflops under optimum conditions now, and the designers are aiming at 1000 megaflops (that is, a billion flops), or speed in *gigaflops*. The Japanese predicted they would have a gigaflop machine by 1990, and some American companies were claiming gigaflop machines in 1989 (e.g., NCUBE and Thinking Machines, according to *Supercomputing Review* (June 1989), pp. 10, 12-13).

Supercomputers make use of what is known as *parallel processing*, in a variety of forms. Instead of being restricted, as earlier machines were, to a strictly *serial* mode of processing information, in which the computer could only perform one action at a time, parallel processing allows several actions to be performed simultaneously. Simpler machine architectures had *one* functional unit in the CPU, the Arithmetic Logical Unit (ALU), which performed all operations; today, machines may have multiple functional units, each of which performs a specific operation or operations. In this way, an "add" functional unit can be doing its job on a pair of operands at the same time that a "multiply" functional unit can be working on another pair of operands.

To speed things up, operations can be overlapped in time; this is generally referred to as *pipelining*. For example, to execute a command involves a four-step process: (1) *get* the instruction from memory; (2) *decode* what instruction is to be performed; (3) *get* the operands from memory; (4) *execute* the operation. Furthermore, step (4) may have several substages, as in a floating-point addition or subtraction, in which the exponents must be compared and made conformable

(by adjusting the smaller operand to have the same exponent as the larger), the add or subtract operation performed, and the result normalized.

If each of these operations is performed in *sequence* for each instruction the machine must perform, many machine clock time periods will be consumed. However, if once an instruction has been started down the "pipeline" past the first step, the next instruction is fed into the pipeline, and so on, matters can be speeded up considerably. This has similarities to a plant assembly line, in which a part moves down the assembly line with each worker on the line performing a small job on it. The whole line does not sit and wait until one part has made the entire trip down the line, but rather parts just keep coming in an (almost) continuous stream, and work proceeds quickly. However, if the job to be done changes, the whole assembly line (or pipeline) operation must be altered. Thus, pipeline overlapping is most useful when the same operation is being performed on many pairs of operands.

Applications of the pipeline technique to *vector processing* (that is, handling operations on one-dimensional or higher-dimensional arrays) are very effective on supercomputers. Let us say, for example, that we are implementing the DO loop:

```
INTEGER A(300), B(300), C(300)
... {arrays A and B are filled somehow}
DO 30 I = 1, 300
    C(I) = A(I) + B(I)
30 CONTINUE
```

If this is done in the normal, sequential manner, it requires many searches for locations A(I) and B(I) as operands, and for C(I) as the place to store the result. Each of these searches involves adding the subscript to the base address for each array, and then accessing the location. The A(2), B(2), C(2) triple access is not done until the A(1), B(1), C(1) stage has been completed. However, if the arrays are handled in a *pipeline* manner, then once A(1), B(1), C(1) have been fed a stage into the pipeline, the next triple can be on its way.

The vectors to be accessed can either be drawn directly from memory, or they may be brought, a segment at a time, into CPU registers, to which the access is much faster. The Cray machines have eight vector registers, each of which can contain up to 64 array elements. On the older versions (Cray-1 and Cray-2), these registers had to be filled from memory one array at a time (since they only had one memory path); but on the Cray X-MP, there are two "fetch" paths and one "store" path to memory, which means that all three arrays in our example could be accessed at the same time (a 64-element segment at a time).

To gain an appreciation of the time saved just in *moving* the data elements ("fetching" two arrays and "storing" into a third), just think about the old accessing method (of having to take the subscript, I, and add it to the base address of the array before determining its location), and compare it with a procedure that will allow you simply to identify a base location (such as the

base address of array A) and say to move a block of 64 elements beginning at that position. Then as the array elements A(I) and B(I) are fed into the pipeline, no new subscript-to-address calculations need to be made; the *next* value for each array from the register is simply fed into the pipeline. Simple and very elegant. There are preprocessors for such machines which can "vectorize" standard FORTRAN programs to run in this manner. However, things will be simpler when Fortran 90 is implemented, at which time we can write our loop (30) simply as:

$$C = A + B$$

and it will be understood to mean adding the entire arrays (and vectorization will be automatic). See Appendix E for more details on the array-handling capabilities of Fortran 90.

There is a standard way of classifying parallel processor machines, which makes reference to their "instruction stream" and their "data stream" (set of operands). A normal mainframe of the kind we are familiar with is called a "single instruction stream, single data stream" (SISD) machine, in which *one* instruction is applied to *one* set of operands at a time. A "single instruction stream, multiple data stream" (SIMD) machine is one with multiple functional units, as we discussed before, so that it can perform (in the different functional units) computations on different data streams. Examples of machines that do this are the Cyber 205 and the various Japanese supercomputers.

A "multiple instruction stream, multiple data stream" (MIMD) machine is one with multiple functional units (so that different operations on different data streams can be executed) and several processors (CPUs), so that multiple programs can be run at the same time, or else a single program can be segmented so that different parts of it are controlled by the different CPUs. An example of this type of machine is the Cray X-MP.

For completeness, the classification scheme allows us to describe a "multiple instruction stream, single data stream" (MISD) machine, but no practical versions of such a machine exist.

There would seem to be a practical, physical limitation on just how fast computers can get. Information is transmitted from one location in the computer to another by electrical signals, and the speed at which these can travel is limited by the speed of light. The speed of light in a vacuum is roughly 186,272 miles per second, which is about .9835 feet per nanosecond (a nanosecond is one billionth of a second, and is a common measure to use for computer speeds these days). Thus if information is to travel from one physical location (say, from memory to a CPU register) to another in the computer in a nanosecond or less, these locations must be less than a foot apart. This would mean the whole system would have to be very small and close-packed to achieve such operation speed. Strides are being made in micro-miniaturization, but again these face physical limitations. Furthermore, the more closely packed the components of the system are, the more concentrated heat they

will generate. This is why mainframes need good air-conditioning systems for their environments. In supercomputers, elements are packed even more closely together, and need super-cooling. An article in *U.S. News and World Report* (July 11, 1983, pp. 46-7) stated that the Cray-1 contained 250,000 memory and logic chips wired together with 70 miles of cable, in a 5-ton machine. The claim was that if the liquid coolant refrigeration system failed, the machine would melt through the floor in a few minutes! The cooling systems currently in use include freon (Cray X-MP), chilled water (IBM), forced air (Fujitsu), and liquid fluorocarbon (Cray-2); the ETA-10 used immersion of the system in liquid nitrogen.

The current major supercomputer manufacturers in this country are Cray (a spinoff of CDC, making the Cray-1, Cray-2, Cray X-MP, and currently developing the Cray-3), Control Data Corporation (the Cyber 205), IBM (the 3090 models), and ETA Systems (a subsidiary of CDC that marketed the Cyber 205s and developed an ETA-10 system, but which CDC has since shut down). The Japanese companies building supercomputers are Hitachi, Fujitsu (allied with Amdahl in the U.S. and Siemens in Europe), and NEC. As was true with mainframes, it is very difficult for a small company (or even a big one) to successfully break into this market against established giants.

In the book, *The Supercomputer Era* (an excellent entree into the field), the authors state (page 172): "The signpost on the road leading toward more productive applications software for supercomputers reads 'FORTRAN'." It is clear that FORTRAN will continue as the major scientific computation language for many years to come, and it (or some modified version) will be the major language of supercomputers (there is also some use of versions of C). The FORTRAN used on supercomputers must be able to take advantage of their parallel processing capabilities. As we mentioned, there are preprocessors for most supercomputer systems which will take a program written in Standard FORTRAN 77 and optimize it to use vectorization, wherever possible, on a supercomputer; some also look for concurrent code. The new Fortran 90 version will include many simplified array manipulation forms that will translate even more readily into vector processing on the supercomputers. However, until that time, it would be a good idea for us to take a brief look at some special considerations that are appropriate in writing FORTRAN programs to run on a supercomputer.

DO loops that reference arrays are the main candidates for vectorization. However, loops with certain characteristics cannot be automatically vectorized by the supercomputer compiler. If you are writing array loops for a supercomputer that you hope will be vectorized, avoid the following in the loop: CALLs or references to subprograms (though many standard FORTRAN functions, such as SQRT, are vectorized); I/O statements; assigned GO TOs; GO TO statements that leave the loop, or that transfer control backward in the loop. Nested IF blocks are best avoided, since they can cause trouble with vectorization (especially if they test on the loop index). Further, any "recur-

sive" array reference, in which an array element calculation depends on the value of a previous element (which might have been changed) cannot be vectorized. Thus, the following DO loops, one that calculates factorials using an array and the second that replaces the values in a filled array with their cumulative sum, could *not* be vectorized:

<pre>***      FACTORIALS      *****       INTEGER FACT(12)       FACT(1) = 1       DO 20 N = 2, 12           FACT(N) = N*FACT(N-1) 20    CONTINUE</pre>	<pre>***      CUMULATIVE SUM  *****       REAL A(100)       READ*, A       DO 30 I = 2, 100           A(I) = A(I)+A(I-1) 30    CONTINUE</pre>
---	---

Some compilers will not vectorize a loop in which there is an assignment to a variable that has been EQUIVALENCEd. Any array appearing in a loop where the array accessing can be identified as beginning at a certain location, ending at a certain location, and proceeding at a fixed "stride" (step size) can be vectorized. Thus, the following loops are vectorizable:

<pre>DO 40 I = 1, 100     B(I) = A(I)**2 - 5 40    CONTINUE</pre>	<pre>DO 50 K = 5, 500, 5     C(K) = B(K/5) 50    CONTINUE</pre>
---	---

Furthermore, references to two-dimensional arrays can be vectorized, because the *stride* between locations is simply the number of rows in the array (due to their column by column arrangement). Similarly, references to arrays of higher dimension can be vectorized in an analogous manner.

Much more detail, and suggestions for writing vectorizable code, can be found in a work such as Levesque and Williamson, *A Guidebook to FORTRAN on Supercomputers*. In addition, use of a Fortran 90 compiler as soon as one becomes available to you will greatly enhance the efficiency of vector processing in your programs on a supercomputer.



## OPTIMIZATION

Optimization is an attempt, by the compiler and/or the programmer, to make as efficient use as possible of the machine hardware available, given the restrictions of the programming language that is being used. Most compilers these days are "optimizing" compilers, and will perform many helpful reinterpretations on translating code that will enhance the program's efficiency. Many compilers have the option that the programmer may specify an optimization *level*—usually the higher the level, the better the optimization (and the longer the compiler takes to perform its job). Since student programs usually contain many errors, and require several compiles, they should gene-

rally be run at a low optimization level. However, if a program is to be put into production use, and most of the bugs have been found and eliminated, a high compiler optimization level should be used, and the object code saved.

Compilers now automatically perform some optimizing steps that used to have to be done by the programmer. For example, in the "old days," a good programmer would realize that an invariant expression should never be recomputed within a loop, and so would take it outside of the loop. For example, the following loop on the left is inefficient because it recalculates  $5.0*A/B$  over and over again in the loop, and an alert programmer in the past would have rewritten it as on the right:

<pre>REAL C(100),D(100),A,B ... DO 60 I = 1, 100     D(I) = 5.0*A/B * C(I) 60 CONTINUE</pre>	<pre>REAL C(100),D(100),A,B ... X = 5.0*A/B DO 70 I = 1, 100     D(I) = X * C(I) 70</pre>
--	---

However, a good modern compiler will recognize the unnecessary redundancy of recalculating  $5.0*A/B$ , and so will calculate it and save it in a register prior to the loop in a more efficient way than the programmer's version on the right. Thus, these days, the good programmer should make it easy for the compiler to recognize invariant expressions, by grouping them together and putting them in parentheses. For example, it would be preferable for the programmer to rewrite the loop expression:

$C(I) = 3.0 + D(I) - A/B$

as

$C(I) = (3.0 - A/B) + D(I)$

to facilitate the compiler recognizing the entire expression in parentheses as removable from the loop.

Similarly, several years ago it would have been desirable for the watchful programmer to rewrite the loop on the left in the form on the right to prevent unnecessary recalculations:

<pre>REAL X(200), Y(200), Z(200) DO 90 I = 1, 200     Z(I) = (X(I) + Y(I)) + &amp;      (X(I) + Y(I))**2 90 CONTINUE</pre>	<pre>REAL X(200), Y(200), Z(200) DO 90 I = 1, 200     T = X(I) + Y(I)     Z(I) = T + T*T 90 CONTINUE</pre>
--	--

However, today's compilers should recognize the re-use of the expression  $(X(I) + Y(I))$ , and calculate and store it more efficiently using a register than the programmer was able to do by creating the temporary variable  $T$ .

There are still a number of things a good programmer should keep in mind that will help in optimizing the program (that is, in making it more efficient).

These considerations should not conflict with writing structured code, but at the same time they can make the program run faster (which is often a real concern in programs of the scope that use supercomputers, or in "real-time" situations such as satellite orbit corrections).

To initialize the values in an array to the same value, or to a set of values that can be grouped, a DATA statement is more efficient than a DO loop (just recall that the DATA statement is not "re-executable," as the DO loop is). Thus, the setup on the right is more efficient than that on the left:

<pre>INTEGER KING(100) DO 20 I = 1, 50     KING(I) = 5     KING(I+50) = 10 20 CONTINUE</pre>	<pre>INTEGER KING(100) DATA KING/ 50*5, 50*10/</pre>
--	--

If loops are nested, and the order of the nesting is not important to the overall effect of the loop, the loop with the most iterations should be made the innermost loop. Compare the two loop structures that follow:

<u>Less Efficient</u> <pre>DO 20 I = 1, 100     DO 20 J = 1, 20         DO 20 K = 1, 5             ... 20 CONTINUE</pre>	<u>More Efficient</u> <pre>DO 20 K = 1, 5     DO 20 J = 1, 20         DO 20 I = 1, 100             ... 20 CONTINUE</pre>
--	--

In the example on the left, the I loop is initialized once, the J loop must be initialized 100 times (once for each I value), and the K loop must be initialized 2000 times (once for each I,J combination, or  $100 \times 20$  times)—a total of 2101 initializations. In the loop structure on the right, however, the K counter is initialized once, the J counter 5 times (once for each value of K), and the I counter is initialized 100 times (once for each K,J combination, so  $5 \times 20$  times)—a total of 106 initializations.

Furthermore, if we look at the number of tests that must be made on completion of the loops, in the example on the left there are 100 tests on completing the outer (I) loop, 2000 tests on the J loop ( $100 \times 20$ ), and 10,000 tests on the K loop ( $100 \times 20 \times 5$ ), a total of 12,100 tests that must be made. In the example on the right, there are 5 tests on completing the outer K loop, 100 tests on completing the J loop ( $5 \times 20$ ), and 10,000 tests on completing the inner (I) loop ( $5 \times 20 \times 100$ ), making a total of 10,105 tests, which is fewer than in the first example.

We can thus see that there are efficiency considerations that indicate we should make the most frequently executed loop the innermost loop, all other things being equal (that is, if this does not change the desired effect of the loop). This approach has an added benefit for supercomputer users. The compilers on the supercomputers generally vectorize only the innermost of a set of nested

loops; thus it is advantageous to make this the loop executed most frequently.

Unnecessary subscript calculations can often be eliminated by making proper use of the step size in the DO loop. Thus, the loop on the right is more efficient than that on the left:

```
DO 10 K = 1, 5
    J = 2*K - 1
    X(J) = X(J)**2
10 CONTINUE
```

```
DO 10 J = 1, 9, 2
    X(J) = X(J)**2
10 CONTINUE
```

A loop may be "unrolled" to do more than one array operation in a loop pass, to reduce loop overhead (initializations and loop termination tests). For example, the example on the left is less efficient than the unrolled loop on the right:

```
REAL A(300), B(300), AVER
DO 20 I = 1, 300
    A(I) = B(I)/AVER
20 CONTINUE
```

```
REAL A(300), B(300), AVER
DO 20 I = 1, 300, 3
    A(I) = B(I)/AVER
    A(I+1) = B(I+1)/AVER
    A(I+2) = B(I+2)/AVER
20 CONTINUE
```

This should not be overdone, however, since it can obscure the readability of the code.

Certainly it should be clear that similar loops on arrays can be combined, as long as one does not depend on the completion of results from the previous loop. Thus, the following example on the right is more efficient (has less loop overhead) than that on the left:

```
REAL A(200), B(200), C(200)
DO 10 I = 1, 200
    A(I) = PI*I**2
    DO 20 J = 1, 200
        B(J) = J + 25
        DO 30 K = 1, 200
            C(K) = K**3 + 2
30 CONTINUE
```

```
REAL A(200), B(200), C(200)
DO 40 I = 1, 200
    A(I) = PI*I**2
    B(I) = I + 25
    C(I) = I**3 + 2
40 CONTINUE
```

If a repetitive operation is involved in a cumulative loop, such as adding a value or multiplying the loop-controlled value by an invariant value, this operation can be saved until the end of the loop to minimize the number of required operations. For example, the loop on the left can be more efficiently replaced by that on the right:

```
REAL A(200), SUM
READ*, A
SUM = 0.
DO 30 I = 1, 200
    SUM = SUM + 2.0*A(I) - 1.
30 CONTINUE
```

```
REAL A(200), SUM
READ*, A
SUM = 0.
DO 40 I = 1, 200
    SUM = SUM + A(I)
40 CONTINUE
SUM = SUM*2.0 - 200.
```

The example on the left requires 200 multiplications, 200 additions, and 200 subtractions; that on the right needs only 200 additions, 1 multiplication, and 1 subtraction (thus resulting in a savings of 398 operations).

Efforts at optimizing your programs will pay off in much improved efficiency, which may be a high priority in certain circumstances. Moreover, if you develop optimizing habits, they will soon become second nature, like using DO loops.



## THE GRAND CHALLENGES IN SCIENCE AND ENGINEERING

A recent report from the President's Office of Science and Technology Policy identified what they called the current Grand Challenges in science and engineering—fundamental problems of great import, the solution of which may be made feasible by the application of “high performance computing resources.” These problems, which surely will be facilitated by the use of supercomputers and effective scientific languages such as FORTRAN, include the following:

**Prediction of weather, climate, and global change.** Understanding the complexities of the biosphere to predict effects of the depletion of the ozone layer, global warming, and the like requires the use of considerable computing power.

**Developments in materials science.** Study of the atomic basis of materials, to bring about advances in superconductors, etc.

**Semiconductor design.** Understanding semiconductor nature.

**Superconductivity;** for energy-efficient power transmission.

**Structural biology.** Study of molecular dynamics by Monte Carlo methods, study of nucleic acids, antibodies, and so on.

**Design of drugs.** Computer simulations to predict the structures of proteins and RNA essential to drug development.

**Human genomes.** Studies of the molecular basis of disease.

**Quantum chromodynamics.** Computer simulations to study the properties of strongly interacting particles, predicting new phenomena, study the conditions of the first microsecond of the “big bang,” and develop unified theories.

**Astronomy.** Large radio telescopes and probes provide quantities of data currently exceeding our computational resources. Expanded computing power could analyze such data.

**Challenges in transportation.** Modeling physical systems to improve fluid flow, aircraft and ship design, and the like.

**Vehicle signatures.** Analysis of characteristics for military purposes (recognition).

**Turbulence.** Study of fluid turbulence in aerospace craft.

**Vehicle dynamics.** Study of stability, performance, ride, and predicted life for land and air vehicles.

**Nuclear fusion.** Its control requires understanding of the behavior of ionized gases at very high temperatures, and the effects of magnetic fields on these systems.

**Efficiency of combustion systems.** Quantum chemistry.

**Enhanced oil and gas recovery.** To locate oil reserves and to procure the oil as economically as possible.

**Computational ocean sciences.** Develop a global ocean prediction model, coupled with atmospheric models.

**Speech.** Spoken language understanding by computers.

**Vision.** Develop vision capabilities of computers.

**Undersea surveillance for anti-submarine warfare.** This was in response to major Soviet developments; effective processing here may require in excess of a trillion computer operations/second.

The government agencies involved in meeting these challenges include DARPA (Defense Advanced Research Projects Agency), NASA, Department of Defense, Department of Energy, NSF, Department of Transportation, National Institute of Health, Health and Human Services, and Office of Naval Research. The source of this information is a government report, "The Federal High Performance Computing Program," Executive Office of the President, Office of Science and Technology Policy (September 8, 1989).

You can see from this report that there is much important work to be done, and it will need people with computational skills such as those you have been developing.



## SUGGESTED READINGS

Arthur, Lowell J. *Programmer Productivity: Myths, Methods, and Murphy's Law*. New York: Wiley, 1984.

Fletcher, R. *Practical Methods of Optimization*. New York: John Wiley, 1987.

- Hayes, John P. *Computer Architecture and Organization*. 2nd ed. New York: McGraw-Hill, 1988.
- Hwang, Kai, and Faye A. Briggs. *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.
- Jones, Gregory. *Software Engineering*. New York: Wiley, 1990.
- Karin, Sidney, and Norris Parker Smith. *The Supercomputer Era*. Boston: Harcourt Brace Jovanovich, 1986.
- Lazou, Christopher. *Supercomputers and Their Use*. Oxford: Clarendon Press, 1986.
- Levesque, John M., and Joel W. Williamson. *A Guidebook to FORTRAN on Supercomputers*. New York: Academic Press, 1989.
- Matsen, F.A., and T. Tajima. *Supercomputers: Algorithms, Architectures, and Scientific Computation*. Austin, Texas: University of Texas Press, 1986.
- Metcalf, Michael. *FORTRAN Optimization*. New York: Academic Press, 1985.
- Ng, Peter A. *Modern Software Engineering: Fundamentals and Current Perspectives*. New York: Van Nostrand Reinhold, 1989.
- Wolfe, Michael. *Optimizing Supercompilers for Supercomputers*. Cambridge, Mass.: MIT Press, 1990.
- Yourdon, Edward. *Managing the Structured Techniques: Strategies for Software Development in the 1990's*. Englewood Cliffs, N.J.: Prentice-Hall, 1985.



## SOURCES OF MATHEMATICAL SUBROUTINE PACKAGES

IMSL (International Mathematical and Statistical Libraries)  
7500 Bellaire Boulevard  
Houston, Texas 77036

NAG (Numerical Algorithms Group)  
1101 31st Street Suite 100  
Downers Grove, Illinois 60515-1263  
Telephone: (312) 917-2337

Wilkinson House  
Jordan Hill Road  
Oxford, United Kingdom OX2 8DR  
Telephone: (0865) 511245

# APPENDIX A



# NUMBER SYSTEMS

*"I have often admired the mystical way of Pythagoras, and the secret magic of numbers."*

- Sir Thomas Browne, Religio Medici

\$\$\$\$\$ There is a certain number of dollar signs preceding this sentence. The number of dollar signs remains the same no matter how we express it. We can give many different names to the number of \$'s shown; we can also use different symbols to represent that number. In English, we say there are "eight" dollar signs; in Spanish, "ochos"; in French, "huit"; in German, "acht"; and so on. We are also familiar with different symbol representations, such as 8 (in Arabic numerals) and VIII (in Roman numerals). We can express the number of dollar signs in various different number systems, systems that are constructed on different number bases. Our representation 8 is in the decimal system, or base 10. However, we could write an expression of the same number in many different number systems.

There is a potential infinity of different possible number systems, but we are practically limited by the availability of different symbols to express counters in the systems. For example, in our decimal system, base 10, we employ ten different counter symbols—0, 1, 2, 3, 4, 5, 6, 7, 8, and 9; in the *binary* (base 2) system, only *two* symbols are employed—0 and 1. The number of dollar signs on our page can be represented variously, in number systems from base 1 to base 9, as:

1111111<sub>1</sub>   1000<sub>2</sub>   22<sub>3</sub>   20<sub>4</sub>   13<sub>5</sub>   12<sub>6</sub>   11<sub>7</sub>   10<sub>8</sub>   8<sub>9</sub>

In a number system in any base b, each position in a symbol string has a particular significance, representing a multiplier times that base b raised to a certain power. The rightmost digit represents a multiplier times the base to the

zero power ( $b^0$ ), or 1; the next position to the left represents a multiplier times that base to the first power ( $b^1$ ); the next position to the left represents a multiplier times the base squared ( $b^2$ ); and so on. Thus, in general, a string of digits of the form  $a_i$  in a base  $b$ ,

$$a_n a_{n-1} a_{n-2} \dots a_3 a_2 a_1 a_0 b$$

represents a value expressed by the following formula:

$$a_n b^n + a_{n-1} b^{n-1} + a_{n-2} b^{n-2} + \dots + a_3 b^3 + a_2 b^2 + a_1 b^1 + a_0$$

For example, when we write  $94782_{10}$  in our customary decimal notation, what we really mean is:

$$9 \times 10^4 + 4 \times 10^3 + 7 \times 10^2 + 8 \times 10^1 + 2$$

or

$$90000 + 4000 + 700 + 80 + 2 .$$

In a course like ours, involving the use of computers, we will be primarily interested in the binary number system (base 2). There are several reasons for this; the main one is that the computer is implemented using elements that have binary forms—switches, bulbs, vacuum tubes, magnetized cores, punched cards, magnetized positions on magnetic tape or disk, and so on. This also connects with the fact that we can use a two-valued system to represent much of our logical reasoning (true or false, in the "Laws of Thought," as described by George Boole in 1854). But we should learn how to convert from one base to another in general.

To convert a string of symbols in some base  $b$  to base 10, we need only to take account of the positional significance of those symbols, as indicated in our earlier formula. Thus, for example,  $314_5 = 3 \times 5^2 + 1 \times 5^1 + 4$ , or  $84_{10}$ , and the value

$$1100101_2 = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^2 + 1, \text{ or } 64+32+4+1, \text{ or } 101_{10}.$$

The conversion from base ten to some other base is somewhat less straightforward, but the method also follows from the formula. The algorithm for conversion to a base  $b$  is to divide through by  $b$ , collecting the remainders, until a quotient of 0 is reached; the remainders are the digits in base  $b$ , from least significant to most significant (that is, the first remainder you obtained after dividing by  $b$  is the  $a_0$  in that base). This follows from the following analysis of the formula:

$$\begin{aligned} & a_n a_{n-1} \dots a_3 a_2 a_1 a_0 b \\ &= a_n b^n + a_{n-1} b^{n-1} + \dots + a_3 b^3 + a_2 b^2 + a_1 b + a_0 \end{aligned}$$

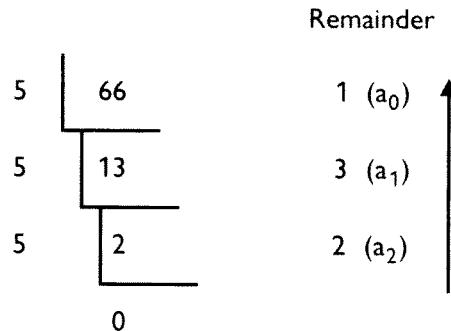
Division by  $b$  gives

$$\div b \quad a_n b^{n-1} + a_{n-1} b^{n-2} + \dots + a_3 b^2 + a_2 b + a_1 \quad \text{rem } a_0$$

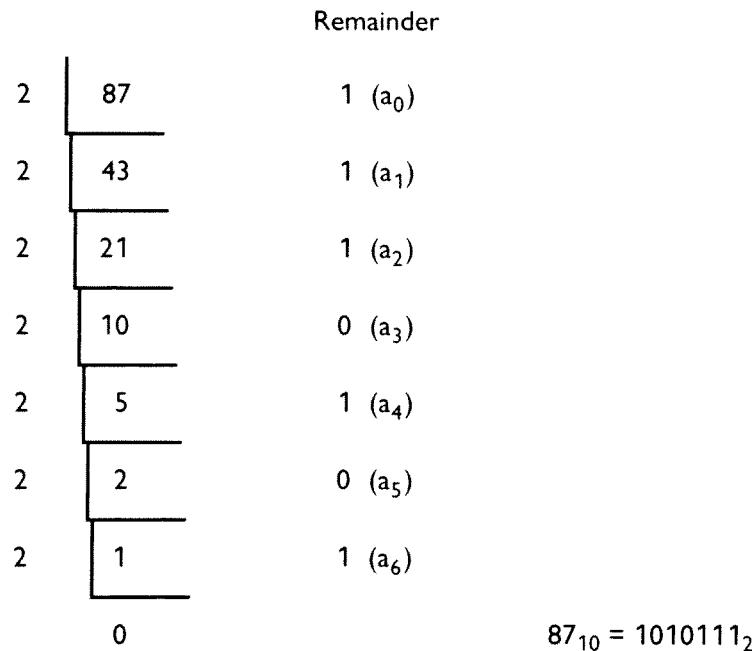
$$\div b \quad a_n b^{n-2} + a_{n-1} b^{n-3} + \dots + a_3 b + a_2 < \text{rem } a_1$$

$$\div b \quad a_n b^{n-3} + a_{n-1} b^{n-4} + \dots + a_3 < \text{rem } a_2$$

and so on, until the resulting quotient is zero. Thus, to convert the base 10 value 66 to base 5, we begin dividing by 5:



so  $66_{10} = 231_5$ . Similarly, to convert  $87_{10}$  to base 2:



Another useful conversion that can be done easily is from base 2 (binary) to base 8 (octal) or to base 16 (hexadecimal). Since 8 is  $2^3$ , one can take every three binary digits (in a whole number, working from right to left, away from the understood binary point) and substituting the equivalent single octal digit. Conversions from octal to binary can be accomplished by replacing each octal digit by its equivalent binary triple. The following table, which counts from 0 to 7 in both bases, displays the appropriate equivalences:

BINARY	OCTAL
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

A similar equivalence table between base 2 and base 16 is:

BINARY	HEXADECIMAL
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Conversions between binary and hexadecimal (base 16) can be similarly performed by equating binary quadruples with single hexadecimal digits, and substituting, since  $16 = 2^4$ .

Thus  $87_{10} = 1010111_2 = 127_8$ , and also  $= 57_{16}$ . To convert in the other direction,  $432_8 = 100011010_2$ , and  $AB_{16} = 10101011_2$ .

## FRACTIONS

A different approach is required to handle fractional values. A fraction in any base  $b$  can be written as a string:

$$0.a_{-1}a_{-2} \dots a_{-m}$$

which means

$$a_{-1}b^{-1} + a_{-2}b^{-2} + \dots + a_{-m}b^{-m}$$

For example, the decimal fraction  $0.435_{10}$  is really

$$4 \times 10^{-1} + 3 \times 10^{-2} + 5 \times 10^{-3} \quad \text{or} \quad 4/10 + 3/100 + 5/1000.$$

Thus, it is not difficult to convert a fraction in some other base to base 10; simply interpret its symbols as multipliers times negative powers of the base, depending on position. So

$$0.34_5 = 3 \times 5^{-1} + 4 \times 5^{-2} = 3/5 + 4/25 = 0.6 + 0.16 = 0.76$$

and

$$0.101_2 = 1 \times 2^{-1} + 1 \times 2^{-3} = 1/2 + 1/8 = 5/8 = 0.625 .$$

Fractions can also be easily converted from binary to octal (or octal to binary), or between binary and hexadecimal. The same technique of substituting equivalent binary triples for single octal digits, or equating a hexadecimal digit with a binary quadruple, working away from the binary point, will work here. Thus, for example

$0.011101_2 = 0.35_8$	$0.0101_2 = 0.24_8$
$0.76_8 = 0.111110_2$	$0.123_8 = 0.001010011_2$
$0.10100001_2 = 0.A1_{16}$	$0.010111111_2 = 0.5FC_{16}$
$0.6F_{16} = 0.01101111_2$	$0.AB4_{16} = 0.101010110100_2$

As with whole numbers, converting a fraction from base ten to some other base is somewhat more complicated; but there is a good algorithm for performing this, based on the formula. If you begin multiplying the fraction in base 10 by the base  $b$  you want to convert to, you will get the values for  $a_{-1}$ ,  $a_{-2}$ , and so on as the whole number parts of the result; then ignore the whole number part and multiply the fractional part by  $b$  again. This operation

is repeated until the fractional part goes to zero (if you are fortunate), or else until you have some desired number of significant fractional digits determined.

$$\begin{aligned}
 & 0.a_{-1}a_{-2}a_{-3} \dots a_{-m} \\
 &= a_{-1}b^{-1} + a_{-2}b^{-2} + a_{-3}b^{-3} + \dots a_{-m}b^{-m} \\
 &\quad \times b \quad \overbrace{a_{-1}}^> + a_{-2}b^{-1} + a_{-3}b^{-2} + \dots a_{-m}b^{-m+1} \\
 &\quad \times b \quad \overbrace{a_{-2}}^> + a_{-3}b^{-1} + \dots a_{-m}b^{-m+2} \\
 &\quad \times b \quad \overbrace{a_{-3}}^> + a_{-4}b^{-1} + \dots a_{-m}b^{-m+3} \\
 \text{etc.}
 \end{aligned}$$

Thus, for example, to convert  $0.44_{10}$  to base 5, multiply:

$$\begin{array}{r}
 0.44 \\
 \times \quad 5 \\
 \hline
 a_{-1} \quad \overbrace{2}^>.20 \\
 \times \quad 5 \\
 \hline
 a_{-2} \quad \overbrace{1}^>.00 \quad \{ \text{stop} \}
 \end{array} \qquad 0.44_{10} = 0.21_5$$

However, some conversions do not terminate nicely as the previous one did. For example, we might want to convert a perfectly innocent-looking base ten value such as  $0.4_{10}$  to binary (to represent, say, 40 cents). We begin multiplying by 2:

$$\begin{array}{r}
 0.4 \\
 \times \quad 2 \\
 \hline
 a_{-1} \quad \overbrace{0}^>.8 \\
 \times \quad 2 \\
 \hline
 a_{-2} \quad \overbrace{1}^>.6 \\
 \times \quad 2 \\
 \hline
 a_{-3} \quad \overbrace{1}^>.2 \\
 \times \quad 2 \\
 \hline
 a_{-4} \quad \overbrace{0}^>.4 \quad 0.4_{10} = \\
 \times \quad 2 \\
 \hline
 a_{-5} \quad \overbrace{0}^>.8 \quad 0.011001100110\dots = \overline{0.0110}_2 \quad \{ \text{a repetition of the first result!} \}
 \end{array}$$

We see that, after the first four terms of the fractional result, the pattern begins to repeat, and so will repeat this way infinitely. Thus we must either decide to just carry out the conversion to some specified number of places, or else find the pattern that repeats, and write a bar over it, implying infinite repetition (as in  $1/3 = 0.333 \dots = 0.\overline{3}$  in base 10).

To convert “mixed” numbers, with both a whole number and a fractional part, simply apply the techniques you have learned separately to the whole number part and the fractional part, and then combine the results.



## BINARY ARITHMETIC

A quick lesson in binary arithmetic will show you how the computer does its work, and how simple it is. In binary addition, there are only four rules to learn:  $0+0=0$ ,  $0+1=1$ ,  $1+0=1$ ,  $1+1=10$ ; if you can add together a pair of numbers, then you can add up a column of numbers (as a sequence of partial sums). Large numbers (more than one bit long) are added right to left (just as you learned in decimal), with carries, if necessary, to the next column. Thus, two examples are illustrated:

Decimal	Binary	Decimal	Binary
13	1101	30	11110
+ 11	1011	+ 15	1111
<hr/> 24	11000	<hr/> 45	101101

Subtraction can be done analogously to that in decimal, by borrowing from the next column if necessary; but actually on the computer it is done as a form of addition. One method to subtract B from A is to add the *one's complement* of B to A. The one's complement of a binary string changes all the zeros to ones and all the ones to zeros (for our purposes, B is considered to have as many bits as A, even if this involves including leading 0s, which will then be complemented to 1s). After the addition, the carry from the leftmost column will not just be written down, but will become an *end-around carry* to the rightmost bit column of the result and added in there. This gives the correct answer.

45	101101	101101	70	1000110	1000110
- 15	- 1111	+ 110000	- 35	- 100011	+ 1011100
<hr/> 30			<hr/> 35		

The multiplication table for binary is also very simple:

$$0 \times 0 = 0 \quad 0 \times 1 = 0 \quad 1 \times 0 = 0 \quad 1 \times 1 = 1$$

Multiplication is performed one digit at a time from the multiplier (just as you learned for decimal, before you got a calculator), and the results are added up (the computer adds up partial sums, a pair at a time, but you may add columns if you like). Notice that multiplying by 7 (111) is just like multiplying first by 1, then by 2 ( $10_2$ ), then by 4 ( $100_2$ ), and that multiplying by a power of two is equivalent to shifting the value multiplied left by a number of places equal to the power, and filling in the 0's (thus multiplying by 8, or  $1000_2$ , or  $2^3$ , is equivalent to shifting the value left three places:  $101_2 \times 1000_2 = 101000_2$ , or  $40_{10}$ ). Thus, the computer performs multiplications as a sequence of shifts and adds (each multiplication is either by a bit value of 0, in which case nothing is added, or by 1, in which case the number being multiplied is just shifted left the appropriate number of places, with zeros filled in, and added).

Decimal	Binary	Decimal	Binary
1 3	1 0 1 1	2 0	1 0 1 0 0
x 7	1 1 1	x 1 2	1 1 0 0
<hr/>	1 1 0 1	<hr/>	0
	1 1 0 1		0
	<hr/>		1 0 1 0 0(0 0)
	1 1 0 1		1 0 1 0 0(0 0 0)
	<hr/>		1 1 1 1 0 0 0 0

Division can be performed, analogously, by shifting and subtracting.

# APPENDIX B



## FORTRAN 77 SYSTEM FUNCTIONS

*"Little by little does the trick."*

- Aesop, "The Crow and the Pitcher," Fables

We give here a list of the system functions generally available on most FORTRAN 77 (and FORTRAN 66) compilers. It is wise to become aware of the availability of such functions, since it will often save you from redoing something that has already been done well.

The type of value a function returns is usually determined by the default type of its name. That is, functions beginning with the letters (I-N) generally return integer values, and functions beginning with (A-H, O-Z) usually return real values, with the following exceptions: several of the system functions beginning with C return complex values, all of the system functions beginning with D return double precision values, and most of the functions beginning with L are the lexical functions to compare character strings with respect to their ASCII ordering, and return logical values (.TRUE. or .FALSE.). Further, FORTRAN 77 introduced the notion of a *generic* function, one which will accept a range of different arguments, and in which the type of the argument often determines the type of the result returned.

We have marked the *generic* functions in the following list with double asterisks (\*\*). They may be used in either the generic or the specific form, except in cases where the function is used as an argument; in such cases, the specific reference should be used (as, for example, when passed to a subprogram).

The type of argument(s) expected by these functions are indicated as follows: X or Y or A(real), K or N (integer), Z (complex), C (character), D (double precision), or G (for arguments to generic functions, which may include integer, real, double precision, and often complex arguments). Gen-

erally, the type of the argument will determine the type of the result the generic function returns, except in cases of type conversion, nearest integer, and the absolute value of a complex argument.

Before the alphabetical listing of system functions, we have given a grouping of function names by type of operation, for your quick reference; the details of a particular function can then be found in the longer, alphabetized list.



## TYPES OF SYSTEM FUNCTIONS

**Absolute Value Functions**—ABS, CABS, DABS, IABS

**Complex Functions**—AIMAG, CABS, CCOS, CEXP, CLOG, CMPLX, CONJG, CSIN, CSQRT

**Double Precision Functions**—DABS, DACOS, DASIN, DATAN, DATAN2, DBLE, DCOS, DCOSH, DDIM, DINT, DLOG, DLOG10, DMAX1, DMIN1, DMOD, DNINT, DPROD, DSIGN, DSIN, DSINH, DSQRT, DTAN, DTANH, IDNINT, SNGL

**Trigonometric Functions**—ACOS, ASIN, ATAN, ATAN2, CCOS, COS, COSH, CSIN, DACOS, DASIN, DATAN, DATAN2, DCOS, DCOSH, DSIN, DSINH, DTAN, DTANH, SIN, SINH, TAN, TANH  
(functions beginning with A refer to the arc-functions, such as arccosine; those ending with H refer to the *hyperbolic* function, such as the hyperbolic tangent—TANH)

**Exponential Functions**—CEXP, DEXP, EXP

**Logarithmic Functions**— ALOG, ALOG10, CLOG, DLOG, DLOG10, LOG, LOG10

**Maximum and Minimum Functions**—AMAX0, AMAX1, AMIN0, AMIN1, DMAX1, DMIN1, MAX, MAX0, MAX1, MIN, MIN0, MIN1

**Type Conversion Functions**—AIMAG, CHAR, CMPLX, DBLE, FLOAT, ICHAR, IDINT, IFIX, INT, REAL, SNGL

**Character Functions**—CHAR, ICHAR, INDEX, LEN, LGE, LGT, LLE, LLT

**Lexical Comparison Functions**—LGE, LGT, LLE, LLT

**Nearest Integer**—IDNINT, NINT

**Nearest Whole Number (as a real)**—ANINT, DNINT

**Positive Difference Functions**—DDIM, DIM, IDIM

**Remainder (Modulo) Functions**—AMOD, DMOD, MOD

**Square Root Functions**—CSQRT, DSQRT, SQRT

**Transfer of Sign Functions**—DSIGN, ISIGN, SIGN

**Truncation Functions**—AINT, DINT


**FORTRAN FUNCTIONS**

\*\* ABS(G) Absolute value of generic argument G  
 ABS(X) Returns absolute value of real argument X  
 \*\* ACOS(G) Arccosine of generic argument G (G in radians)  
 ACOS(X) [or ARCOS(X)] Arccosine of X (in radians)  
 AIMAG(Z) Returns imaginary part of argument Z  
 \*\* AINT(G) Truncates generic argument G  
 AINT(X) Integer part (largest integer  $\leq |X|$ )  
 ALOG(X) Natural logarithm of X  
 ALOG10(X) Logarithm to the base 10 of real argument X  
 AMAX0(N1,N2, . . .) Returns (as a real) the maximum value of  
     two or more integer arguments  
 AMAX1(X1,X2, . . .) Returns the real maximum value of two or  
     more real arguments  
 AMIN0(N1,N2, . . .) Returns (as a real) the minimum value of  
     two or more integer arguments  
 AMIN1(X1,X2, . . .) Returns real minimum value of two or more  
     real arguments  
 AMOD(X,A) Remainder when real X is divided by real A  
 \*\* ANINT(G) Nearest whole number to generic argument G  
 ANINT(X) Nearest whole number to real X  
 \*\* ASIN(G) Arcsine of generic argument G (G in radians)  
 ASIN(X) [or ARSIN(X)] Arcsine of real X (in radians)  
 \*\* ATAN(G) Arctangent of generic argument G (G in radians)  
 ATAN(X) Arctangent of real X (in radians)  
 \*\* ATAN2(G1,G2) Arctangent of G1/G2 (G1, G2 are generic)  
 ATAN2(X1,X2) Arctangent of X1/X2 (X1, X2 are reals)  
 CABS(Z) Real absolute value of complex number Z  
 CCOS(Z) Complex cosine of complex argument Z (radians)  
 CEXP(Z) Exponential:  $e^{**Z}$  (where Z is complex)  
 CHAR(N) Returns the character in the Nth position of the  
     processor collating sequence  
 CLOG(Z) Complex natural logarithm of complex Z  
 \*\* CMPLX(G1, G2) Returns complex number  $G1 + iG2$   
     (G2 optional)  
 CMPLX(X,Y) Returns complex number  $X + iY$  (Y is optional)  
 CONJG(Z) Complex conjugate of Z; if  $Z = a + ib$ ,  $\bar{Z} = a - ib$   
 \*\* COS(G) Trigonometric cosine of G (G in radians)  
 COS(X) Trigonometric cosine of X (in radians)

\*\* COSH(G) Hyperbolic cosine of G (G in radians)  
 COSH(X) Hyperbolic cosine of X (in radians)  
 CSIN(Z) Complex sine of Z (in radians)  
 CSQRT(Z) Complex square root  
 DABS(D) Double-precision absolute value  
 DACOS(D) [or DARCOS(D)] Double-precision arccosine of D  
     (D is expressed in radians)  
 DASIN(D) [or DARSIN(D)] Double-precision arcsin of D radians  
 DATAN(D) Double-precision arctangent of D (in radians)  
 DATAN2(D1,D2) Double-precision arctangent of D1/D2  
 \*\* DBLE(G) Type conversion of G to double-precision  
 DBLE(X) Type conversion of X from real to double-precision  
 DCOS(D) Double-precision cosine of D radians  
 DCOSH(D) Double-precision hyperbolic cosine of D radians  
 DDIM(D1,D2) Double-precision positive difference (see DIM)  
 DEXP(D) Double-precision exponential:  $e^{**D}$   
 \*\* DIM(G1,G2) Positive difference of arguments G1 and G2  
 DIM(X,Y) Positive difference of reals X and Y: if  $X > Y$ ,  
     returns  $X - Y$ ; if  $X \leq Y$ , returns 0  
 DINT(D) If  $D < 1$ , returns 0; if  $|D| > 1$ , returns the integer whose  
     magnitude is the largest integer that does not exceed D, and whose  
     sign is the same as D  
 DLOG(D) Double-precision natural logarithm  
 DLOG10(D) Double-precision base 10 logarithm of D  
 DMAX1(D1,D2, . . .) Double-precision maximum of two or more  
     double precision arguments  
 DMIN1(D1,D2, . . .) Double-precision minimum of two or more  
     double-precision arguments  
 DMOD(D1,D2) Double-precision remainder when D1 is divided by D2  
 DNINT(D) Nearest whole number (in double precision) to D  
 DPRD(X,Y) Double-precision product  $X * Y$   
 DSIGN(D1,D2) Transfer of sign of D2 to d.p. argument D1  
 DSIN(D) Double-precision sine of D radians  
 DSINH(D) Double-precision hyperbolic sine of d.p. D  
 DSQRT(D) Double-precision square root  
 DTAN(D) Double-precision tangent  
 DTANH(D) Double-precision hyperbolic tangent  
 \*\* EXP(G) Exponential:  $e^{**G}$   
 EXP(X) Exponential:  $e^{**X}$   
 FLOAT(N) Returns real equivalent value of integer N

IABS(N) Integer absolute value of integer argument N  
 ICHAR(C) Position of character C in system collating sequence  
 IDIM(N,K) Positive difference of N - K; if N > K, = N - K;  
     if N <= K, returns 0  
 IDINT(D) Type conversion of double-precision argument to its  
     integer part  
 IDNINT(D) Nearest integer to double-precision value D  
 IFIX(X) Type conversion, real X to integer  
 INDEX(C1,C2) Position within string C1 of substring C2;  
     if C2 is not in C1, returns 0  
 \*\* INT(G) Integer part of generic argument G  
 INT(X) Integer part of real X (largest integer  $\leq |X|$ )  
 ISIGN(N,K) Transfer of sign of K to value of N  
 LEN(C) Length of character string C  
 LGE(C1,C2) Returns TRUE of C1  $\geq$  C2 (i.e., same or follows) in  
     ASCII collating sequence, FALSE otherwise<sup>1</sup>  
 LGT(C1,C2) Returns TRUE if C1 > C2 (i.e., follows it) in the  
     ASCII collating sequence, FALSE otherwise<sup>1</sup>  
 LLE(C1,C2) Returns TRUE if C1  $\leq$  C2 (i.e., same or precedes) in  
     ASCII collating sequence, FALSE otherwise<sup>1</sup>  
 LLT(C1,C2) Returns TRUE if C1 < C2 (i.e., precedes it) in the  
     ASCII collating sequence, FALSE otherwise<sup>1</sup>  
 \*\* LOG(G) Natural logarithm of argument G  
 \*\* LOG10(G) Common logarithm (base 10) of argument G  
 \*\* MAX(G1,G2,...) Maximum value of two or more generic  
     arguments  
 MAX0(N1,N2,...) Maximum value of two or more integer  
     arguments  
 MAX1(X1,X2,...) Maximum value (as an integer) of two or  
     more real arguments  
 \*\* MIN(G1,G2,...) Minimum value of two or more generic  
     arguments  
 MIN0(N1,N2,...) Minimum value of two or more integer  
     arguments

---

<sup>1</sup>Note: The LGE, LGT, LLE, and LLT functions compare two character string arguments regarding their *lexical* ordering in the ASCII collating sequence. If you use normal logical relations (.GE. and so on) to examine two character string arguments, you will get a logical value TRUE or FALSE depending on their order in the collating sequence of the processor you are using.

MIN1(X1,X2, . . .) Minimum value (as an integer) of two or more real arguments  
\*\* MOD(G1,G2) Remainder when G1 is divided by G2  
MOD(N,K) Remainder when N is divided by K (N modulo K)  
\*\* NINT(G) Nearest integer to generic argument G  
NINT(X) Nearest integer to real value X (rounds)  
\*\* REAL(G) Type conversion of argument G to real  
REAL(N) Type conversion of integer argument N to real  
\*\* SIGN(G1,G2) Transfer of sign of argument G2 to G1  
SIGN(X,Y) Transfer of sign of real Y to real argument X  
\*\* SIN(G) Trigonometric sine of argument G (G in radians)  
SIN(X) Trigonometric sine of X (X is angle in radians)  
\*\* SINH(G) Hyperbolic sine of G (G in radians)  
SINH(X) Hyperbolic sine of X (in radians)  
SNGL(D) Converts double-precision argument D to single precision  
\*\* SQRT(G) Square root of generic argument G  
SQRT(X) Square root (positive) of positive real arg. X  
\*\* TAN(G) Trigonometric tangent of argument G (radians)  
TAN(X) Trigonometric tangent of X (X is in radians)  
\*\* TANH(G) Hyperbolic tangent of generic argument G  
TANH(X) Hyperbolic tangent of X (in radians)

---

Fortran 90 has added a number of new functions to this collection, primarily functions for handling arrays, for bit manipulation, and for determining parameters of the system being used. Coverage of these functions is included in Appendix E.

# APPENDIX C



## CAPSULE SUMMARY OF FORTRAN 77 STATEMENTS

*"Everyone must row with the oars he has."*

*English Proverb*

Generally, the available FORTRAN statements or statement types are listed here alphabetically, but a few are listed under their more general heading. Thus file handling instructions, such as BACKSPACE, OPEN, CLOSE, and so on are listed under "File Handling," and the various type specification statements such as INTEGER, CHARACTER, and IMPLICIT are listed under "Type Statements."

We must begin by discussing the form, or layout, of a line in FORTRAN, since all instructions in FORTRAN must follow this specification. Columns 1–5 are reserved for statement numbers, if they are used; thus statement numbers may range from 1 to 99999. The one exception to this is a comment line, which must have a C or an asterisk (\*) in column 1; then the rest of that line will be treated as a comment, and ignored by the compiler. Column 6 is for a continuation symbol (anything but a blank or 0), which indicates that the line is a continuation of the line before it. Columns 7–72 are for the FORTRAN instruction. Columns 73–80 are ignored, and might be used for sequencing.

The kinds of declaration statements, expressions, control structures, and commands available in FORTRAN are:

**Assignment Statements**—An assignment statement is of the form:

variable = expression

which evaluates the expression on the right and stores its result into the variable named on the left. The expression must be of the appropriate type to be stored in the variable. Variable names may be one to six characters long, must begin with a letter, and may be made up of letters and/or digits. Variables may be of different types (as may the constants that are stored in them)—

integer, real, complex, double precision, logical, or character. The sorts of operations allowed in an expression depend upon the types of values involved, as follows:

*Arithmetic expressions*—These may perform any of the standard arithmetic operations, including exponentiation (\*\*). The operators are executed in hierarchical order, as follows:

( ), \*\*, \* or /, + or -

If the hierarchy does not resolve the order, they are evaluated from left to right (except for successive exponentiations, which are evaluated from right to left, as in A\*\*B\*\*C).

*Character expressions*—Character values may be compared as to their lexical ordering in logical expressions (q.v.) or by logical lexical functions (see Appendix B). The only operation that can be performed on characters is that of *concatenation*—that is, stringing them together with the operator //, such as:

```
C = 'A'//' GOOD'//' DAY'
```

which stores 'A GOOD DAY' in C, if all the characters will fit.

*Logical expressions*—These compare two values with respect to magnitude (or lexical order), and are the logical relations:

.LT. (<), .LE. (<=), .GT. (>), .GE. (>=), .EQ. (==), .NE. (/=)

Logical expressions may be combined using the logical operators such as .AND., .OR., .NOT., .EQV., or .NEQV. as in the following:

```
((A.GT.B).OR..NOT.(X.LT.Y.AND.X.LE.Z))
```

The precedence of logicals is first all logical relations, then .NOT., then .AND., then .OR., and finally .EQV. and .NEQV.

**ASSIGN Statement**—The ASSIGN statement is of the form:

ASSIGN statement label TO integer variable

It gives the value of some statement number in the program to the integer variable named in the expression, and that variable may then be referenced in an Assigned GO TO statement (q.v.).

**BLOCK DATA**—This statement represents the entry line to a special subprogram, in which DATA statements may be used to assign values to variables in labelled COMMON (see these terms).

**CALL**—The CALL statement appears in a program unit as a reference to a subroutine, causing entry to that subroutine. The CALL is followed by the name of the subroutine (see SUBROUTINE).

**Comments**—A comment or remark may be inserted into the body of a program by placing a C or \* in column 1; the rest of the line is ignored by the compiler, but appears on any program listing.

**COMMON**—The COMMON statement declares any variables following it to be located in a special area of memory, one that may be accessed by any other subprogram containing a similar COMMON statement. It allows sharing of variables among program units. Variables may be stored in either blank or labelled COMMON, where the label appears between slashes. Examples of both follow:

```
COMMON A, B, C /AREA1/HATE, LOVE
```

Variables A, B, and C are in blank COMMON, in that order, and the variables HATE and LOVE are in the labelled area called AREA1. The corresponding variable names in another program unit need not match those in the first, since COMMON refers to locations in the order specified in the COMMON statement. Thus, in another unit,

```
COMMON X, Y, Z
```

will refer to the same three locations as A, B, and C in the first. However, labels of areas must always be the same (e.g., /AREA1/).

**CONTINUE**—A “do-nothing” statement that sends the program on to the next executable step in the program. This statement is generally used to mark the terminal statement of a DO loop (q.v.).

**DATA statement**—A DATA statement is of the general form:

```
DATA varlist1/constantlist1/[,varlist/constlist/]1
```

and is used to assign values to variables *at compile time*. Any variable or array may be initialized, except for dummy arguments and variables in blank COMMON. The constant list must agree in length (and generally should agree in type) with the variable list. Array locations may be specified in implied lists, as in:

```
INTEGER MAT(50), CAT(20)
DATA (MAT(I), I = 1, 50)/50*4/, CAT/10*5,10*3/
```

and a repetition factor followed by an asterisk (\*) may be used in the constant

---

<sup>1</sup>Note: Sections of a command format set off in square brackets (that is, in [ ]) indicate parts of the command that are *optional*, and may be used or omitted as appropriate.

list. The reference to CAT, above, without subscripts, indicates the whole array, in order.

**DIMENSION**—This statement is used to allocate memory for arrays. Such arrays may be from one to seven dimensions in FORTRAN 77. The DIMENSION statement names each array, with its dimension parameter(s) in parentheses. Each dimension parameter may be a single integer value, or two integers specifying a range, e.g.:

```
DIMENSION A(40), B(-5:5), C(10, 21:40)
```

The array A will contain 40 locations, named A(1) through A(40); the array B will contain 11 locations, named B(-5), B(-4), . . . , B(0), B(1), . . . , B(5); and array C will have 10 rows (1 through 10) and 20 columns (numbered 21 through 40). An array may alternatively be dimensioned in its type declaration statement.

**DO loop**—A DO loop is initialized by a statement of the form:

```
DO n loopvar = init, final [,step]
```

where n is a statement number following the DO, and indicates that the loop includes all statements from the one immediately following the DO through the one numbered n. The loop variable controls the execution of repetitions of the loop from init to final in steps of step. If step is not specified, it is assumed to be +1. The terminal statement (numbered n) of the loop may be any executable statement except a GO TO, STOP, RETURN, ELSE, ELSEIF, ENDIF, DO, IF/THEN, or arithmetic IF; often it is a CONTINUE statement.

**END**—The END statement is the last physical statement in a program unit (required), which indicates the end of statements to be compiled.

**ELSE, ELSEIF, ENDIF**—See Block IFs under IF Statements.

**ENTRY**—The ENTRY statement is optional in a subprogram, indicating an alternate entry point to the code in the subprogram. The entry line must specify a name, and may also specify arguments:

```
SUBROUTINE GOOD(A, B, C)
      .
      .
      .
      ENTRY BAD (X, Y)
      .
      .
      .
      ENTRY UGLY
```

If the subprogram is referenced by one of its alternate entry names (e.g., CALL UGLY), execution will begin at the first statement following the entry point.

**EQUIVALENCE**—The EQUIVALENCE statement allows specification of several alternate names for the same location(s), used to patch up errors, save memory space, or allow specification of location contents using more than one type of reference. It is of the form:

EQUIVALENCE (var<sub>1</sub>, var<sub>2</sub> [.var]) [, (vara, varb . . .)]

For example, the following statements:

```
LOGICAL X(1000)
DIMENSION A(20,50), NAT(10,10,10)
EQUIVALENCE (X(1), A(1,1), NAT(1,1,1))
```

give 1000 locations three alternate structures and types.

**EXTERNAL**—The EXTERNAL statement allows the programmer to declare user-defined subprograms as “externals” so that they may be passed as arguments to other subprograms. See also INTRINSIC.

**FORMAT**—A Format is a numbered statement, referred to by a PRINT, WRITE, or READ statement, which describes the layout of the data to be output or input. The components of a FORMAT statement include a single character at the beginning of an output format which determines “carriage control”: ‘1’ for top of new page, ‘0’ for skip a line, ‘ ’ for next line, and ‘+’ for same line. The other edit descriptors are nX (for n spaces), Tn (tab to position n), Iw (for integers), Fw.d or Ew.d or Gw.d (for reals), Dw.d (for double precision), Lw (for logicals), and A or Aw (for characters), where w indicates the field width of the item, and d the number of decimal places to the right of the decimal point. A repetition factor may be used before any edit descriptor (e.g., 5I4). A slash (/) at the beginning or end of a FORMAT statement indicates a skipped record; a single slash in the middle of a FORMAT indicates the end of the record described to its left; additional slashes indicate skipped records.

**FUNCTION**—a function subprogram has an entry line of the form:

FUNCTION name (arguments)

and is compiled as a separate program unit. Its dummy arguments must agree in number and type with actual arguments passed to the function. At least one line in the function should define a value for the function name to return. A function should contain at least one RETURN statement (though an END will substitute). The primary purpose of a function is to calculate *one* value.

A *Statement Function* is a one-liner of the form:

fnname(args) = expression(args)

which defines some expression, under a specific name, for use in the program unit in which it appears.

**GO TO**—The GO TO is an unconditional branch statement. In FORTRAN 77 there are three types of GO TOs. The simplest is of the form:

GO TO n

where n is a statement number in the program, and indicates a branch to the instruction having that statement number.

An ASSIGNED GO TO is of the form.

GO TO intvar

where a value has been given to intvar by an ASSIGN statement (q.v.).

A Computed GO TO is of the form (where *int* has a value 1-m):

GO TO (n<sub>1</sub>, n<sub>2</sub>, ..., n<sub>m</sub>)[,] int

The parentheses enclose a list of statement numbers which appear in the program. If the value of *int* (an integer variable or expression) is 1, the branch is taken to n<sub>1</sub>, if it is 2, the branch is taken to n<sub>2</sub>, and so on. If *int* does not evaluate to a value between 1 and m, the statement following the GO TO is executed.

**File Handling**—The use of external files by the program involves a number of special instructions, which are detailed in Chapter 11. These instructions include the following:

BACKSPACE u—positions at the beginning of preceding record

CLOSE (clist)—disconnects the particular file specified

ENDFILE u—writes an end-of-file record on the file at unit u

INQUIRE (clist)—determines properties of a specified file or the connection to a specified unit; variables return values

OPEN (clist)—connects a specified file so that it may be accessed by the program for input or output

REWIND u—positions at the beginning of the file on unit u

The *clist* specification contains file and unit definitions, as well as a number of parameters or variables through which information about the status of the operation can be returned.

**IF Statements**—These are the *conditional branches* in FORTRAN. There are basically three sorts of IF statements in the language:

The Arithmetic IF is of the form:

IF (arith.exp.) n<sub>-</sub>, n<sub>0</sub>, n<sub>+</sub>

The arithmetic expression in parentheses is evaluated—if it is negative, the branch to statement  $n_-$  is taken; if it is zero, the branch to  $n_0$  is taken; if it is positive, it branches to  $n_+$ .

The Logical IF is of the form:

```
IF (logical exp.) executable statement
```

The logical expression (see forms under Assignment Statements) in parentheses is evaluated; if it is True, the executable statement following is executed; if it is not true, the program drops down to execute the next statement in the sequence.

The Block IF is of the form:

```
IF (logical exp.) THEN
    block of executable statements
ENDIF
```

in its simplest form. If the logical expression in parentheses is True, the following block of statements is executed; if not, it is skipped. A Block IF can also execute one block if a condition is True, another if it is False:

```
IF (logical exp.) THEN
    Block of st's executed if condition True
ELSE
    Block executed if condition is False
ENDIF
```

One further elaboration on the Block IF allows for many tests:

```
IF (logical exp1) THEN
    Block1 (if exp1 is True)
ELSEIF (logical exp2) THEN
    Block2
ELSEIF (logical exp3) THEN
    Block3
.....
[ELSE
    All-else-fails block ]
ENDIF
```

**INTRINSIC**—This statement allows the programmer to declare certain system functions as “intrinsics” so that they may be passed as arguments to other subprograms. See also EXTERNAL.

**PARAMETER**—This statement allows you to define symbolic names for constants in your program. These names can then be used in any place

where the constant would have appeared, and the system prevents their values from being changed. It is of the form:

PARAMETER (par = expr [,par = expr])

where par is a symbolic name, and expr is any arithmetic expression in constants or previously defined parameters.

**PAUSE**—This statement causes an interruption in the execution of the program at that point. The program may then be resumed by an action from outside the program. The statement may simply be PAUSE, or PAUSE n, where n is a string of up to 5 digits, or a character constant, which is accessible at the time the program is paused. This feature is rarely used nowadays.

**PRINT (or WRITE)**—This is an output statement, usually in conjunction with a FORMAT to describe the form of the output. It may be in the form PRINT\* [, list] (no format used), or PRINT n [,list] (n refers to the statement number of the format), or it may be WRITE (u,n) [list] (where u is the unit number to be written to), or WRITE (\*,n) [list] or WRITE (u,\*)[list] or WRITE (\*,\*)[list]. The list of variables or expressions to be written is optional. If an \* is used for the output unit number, it is the default output device.

**PROGRAM**—This is an optional statement at the beginning of the program, of the form PROGRAM name. If used, it must appear first in the program. The *name* must follow normal naming conventions and not be the same as any local variable name, COMMON block, or external.

**READ**—This is the input statement for a FORTRAN program, which may use a FORMAT statement number *n* or not (\*), and may specify a unit number *u* to be read from, or the default input device (\*). It may be of the form READ n, list, or READ\*, list, or READ(u,n) list, or READ(\*,n) list, or READ(u,\*) list, or READ (\*,\*) list. The READ statement may optionally include an END= or ERR= clause:

READ (u, n, END= sn1, ERR= sn2)

The END= and ERR= clauses allow the programmer to indicate statement numbers (sn's) to be branched to on the condition that an end-of-file is encountered on the read (END = sn), or if an error condition occurs (ERR= sn).

**RETURN**—This is an optional statement (if it is omitted, the END statement will be taken to imply return) in a subprogram unit, which indicates the point at which transfer of control is to be returned to the calling program unit.

*Alternate RETURNS* may be specified to a subroutine (not a function) by including asterisks (\*'s) {or on some systems, \$'s} in the parameter sequence in the entry line, and supplying corresponding statement numbers preceded by \*'s (or \$'s) in the CALL statement which indicate the different statements the RETURN should come to on re-entering the calling program. Which RETURN is actually taken is then determined by the execution of a RETURN *n* statement in the subroutine, where *n* is an integer that tells which of the starred statements (first, second, etc.) to use; if no integer follows the RETURN executed, it is a normal return to the first executable statement following the CALL. For example,

```
CALL ROOT (A, N, *88, *99, *77)
SUBROUTINE ROOT (X, K, *, *, *)
```

Then in subroutine ROOT, if RETURN 1 is executed, it will pick up at statement number 88 in the calling program; if RETURN 2 is executed, the return will be to statement 99, and if RETURN 3 is executed, it will pick up at statement 77; if just a plain RETURN is executed, the statement following the CALL will be executed.

**STOP**—This statement may be included in a program unit. Its execution causes termination of the program.

**SUBROUTINE**—A Subroutine statement defines the entry line of a subroutine subprogram, with a name, and which may or may not have an argument list:

```
SUBROUTINE TOPS      or      SUBROUTINE GOOD (A, N, K, BAD)
```

The argument list contains dummy arguments that get their identity on a program CALL to the subroutine CALL name [(args)]. It should contain at least one RETURN statement (though the END may be used instead), and must have an END statement. It may also make use of COMMON statements for sharing variables.

**Type Statements**—These statements declare a list of variables to be given particular types in the program. They may be INTEGER, REAL, COMPLEX, DOUBLE PRECISION, LOGICAL, CHARACTER, or IMPLICIT. A CHARACTER declaration must also specify the lengths of character variables longer than one character. An IMPLICIT statement may declare a whole range of variables beginning with certain letters to be of a particular type or types. For example,

```
IMPLICIT INTEGER (A-C,S), REAL (M), LOGICAL (X-Z)
CHARACTER A*4, B*7, C*6
CHARACTER*3 FAT, CAT, RATS*8
```

All variables not explicitly typed in such statements will follow the default naming convention in FORTRAN—those beginning with I through N are integer, and the remainder are reals.

**WRITE**—See **PRINT**.

# APPENDIX D



## ASCII AND EBCDIC CODING SYSTEMS

*"I had also invented a set of symbols for the typewriter like FORTRAN has to do, so I could type equations."*

- Richard P. Feynman, "Surely You're Joking, Mr. Feynman!":  
Adventures of a Curious Character, p. 12

There have been several representation schemes for coding data internally in the computer, but ASCII and EBCDIC are the most widely used. ASCII, which stands for American Standard Coding for Information Interchange, is basically a 7-bit code, which allows for the encoding of 128 different symbols. However, the "byte" containing an ASCII character code is generally an 8-bit byte, with the eighth bit used as a parity (check) bit or other purpose. It was intended to serve as an international standard, and it is the most widely used coding scheme. However, there are other schemes, so we cannot consider that ASCII acts as the standard at this time.

ASCII CODES FOR PRINTABLE CHARACTERS

Character	Octal	Hex	Character	Octal	Hex
blank	040	20	P	120	50
!	041	21	Q	121	51
"	042	22	R	122	52
#	043	23	S	123	53
\$	044	24	T	124	54

Character	Octal	Hex	Character	Octal	Hex
%	045	25	U	125	55
&	046	26	V	126	56
'	047	27	W	127	57
(	050	28	X	130	58
)	051	29	Y	131	59
*	052	2A	Z	132	5A
+	053	2B	[	133	5B
,	054	2C	\	134	5C
-	055	2D	]	135	5D
.	056	2E	^	136	5E
/	057	2F	_	137	5F
0	060	30	'	140	60
1	061	31	a	141	61
2	062	32	b	142	62
3	063	33	c	143	63
4	064	34	d	144	64
5	065	35	e	145	65
6	066	36	f	146	66
7	067	37	g	147	67
8	070	38	h	150	68
9	071	39	i	151	69
:	072	3A	j	152	6A
;	073	3B	k	153	6B
<	074	3C	l	154	6C
=	075	3D	m	155	6D
>	076	3E	n	156	6E
?	077	3F	o	157	6F
@	100	40	p	160	70

Character	Octal	Hex	Character	Octal	Hex
A	101	41	q	161	71
B	102	42	r	162	72
C	103	43	s	163	73
D	104	44	t	164	74
E	105	45	u	165	75
F	106	46	v	166	76
G	107	47	w	167	77
H	110	48	x	170	78
I	111	49	y	171	79
J	112	4A	z	172	7A
K	113	4B	{	173	7B
L	114	4C		174	7C
M	115	4D	}	175	7D
N	116	4E	~	176	7E
O	117	4F			

*Note:* Codes 00-31 and 7F (in hex) are unprintable control characters.

The next coding scheme we will describe is EBCDIC (Extended Binary Coded Decimal Interchange Code). This is used on some Burroughs and IBM mainframes, as well as by Hewlett-Packard and Amdahl. In this coding, the special characters are numerically less than the lowercase letters (a-z), the lowercase letters are less than the uppercase letters (A-Z), and the uppercase letters are less than the digits (0-9). EBCDIC is an 8-bit code, so there are  $2^8$ , or 256, possible codings. Some are used for control characters, and some are not utilized at all. We present the printable character codes here.

If your system allows the printing of CHARACTER variables with O (Octal) or Z (hexadecimal) format, you can view these coding schemes first-hand. Usually, when characters are stored in appropriate-sized bytes on a computer, the character string is stored left-adjusted in the character space made available by the type statement, and blank-filled if the string is shorter than the space allowed. For example, if you have the type declaration

```
CHARACTER C*6
```

and you store 'CAT' in C, it will be left-adjusted and blank filled, as indicated:



and as many characters as are asked for are taken, from the left, so that printing C with an A1 format gives C, A2 gives CA, A3 gives CAT, A4 gives CAT , and so on.

EBCDIC CODES FOR PRINTABLE CHARACTERS

Character	Binary	Hex	Character	Binary	Hex
blank	01000000	40	~	10100001	A1
¢ or [	01001010	4A	s	10100010	A2
.	01001011	4B	t	10100011	A3
<	01001100	4C	u	10100100	A4
(	01001101	4D	v	10100101	A5
+	01001110	4E	w	10100110	A6
	01001111	4F	x	10100111	A7
&	01010000	50	y	10101000	A8
! or ]	01011010	5A	z	10101001	A9
\$	01011011	5B	A	11000001	C1
*	01011100	5C	B	11000010	C2
)	01011101	5D	C	11000011	C3
;	01011110	5E	D	11000100	C4
—	01011111	5F	E	11000101	C5
—	01100000	60	F	11000110	C6
/	01100001	61	G	11000111	C7
,	01101011	6B	H	11001000	C8
%	01101100	6C	I	11001001	C9
_	01101101	6D	J	11010001	D1
>	01101110	6E	K	11010010	D2

Character	Binary	Hex	Character	Binary	Hex
?	01101111	6F	L	11010011	D3
:	01111010	7A	M	11010100	D4
#	01111011	7B	N	11010101	D5
@	01111100	7C	O	11010110	D6
,	01111101	7D	P	11010111	D7
=	01111110	7E	Q	11011000	D8
"	01111111	7F	R	11011001	D9
a	10000001	81	\	11100000	E0
b	10000010	82	S	11100010	E2
c	10000011	83	T	11100011	E3
d	10000100	84	U	11100100	E4
e	10000101	85	V	11100101	E5
f	10000110	86	W	11100110	E6
g	10000111	87	X	11100111	E7
h	10001000	88	Y	11101000	E8
i	10001001	89	Z	11101001	E9
j	10010001	91	0	11110000	F0
k	10010010	92	1	11110001	F1
l	10010011	93	2	11110010	F2
m	10010100	94	3	11110011	F3
n	10010101	95	4	11110100	F4
o	10010110	96	5	11110101	F5
p	10010111	97	6	11110110	F6
q	10011000	98	7	11110111	F7
r	10011001	99	8	11111000	F8
			9	11111001	F9

EBCDIC codes on some machines may differ slightly from this, since the coding scheme is not standardized.

## APPENDIX E



# FEATURES OF FORTRAN 90

*"Curiouser and curiouser."*

- Lewis Carroll, Alice in Wonderland

As soon as the FORTRAN 77 Standard came out in 1978, a group was formed to look into what the next version of FORTRAN should be like. This group, the X3J3 Technical Committee, reports to ANSI (the American National Standards Institute), and over the past decade has been evaluating various proposals to "modernize" FORTRAN. The references made to the next FORTRAN version, referred to as Fortran 90, throughout this book, are based on the final draft report (S8, Version 118, May 1991) from this Committee. This report has already been accepted as the international standard for Fortran 90, and is still under review at this writing in the United States. Thus, the final form of the new language version may differ slightly from what we have discussed here, but such differences are likely to be minor. Earlier versions of the revision called the proposal Fortran 8x, so you may occasionally see references to this name.

When the new Standard is accepted, compiler writers will then have to provide actual software that embodies the Standard. This procedure will take time, as will the acquisition and adoption of a new compiler at various installations. If we look at the period of time over which FORTRAN 77 adoption took place, we see that it can take a number of years. We also note that FORTRAN '66 compilers remained available in some places for many years after the introduction of FORTRAN 77, a tribute to the longevity of that earlier version.

Thus, we anticipate that the production and adoption of Fortran 90 will take several years, and that FORTRAN 77 will continue to be actively used for some years after that. We have summarized the more important features of Fortran 90 that will impact the scientific programmer for the information of the reader, who should then be well prepared to make the transition when Fortran 90 does become generally available.

The overview of the May 1991 document indicates the seven major additions to FORTRAN 77 that this new version will provide:

1. Array operations
2. Improved facilities for numeric computation
3. Extended intrinsic data types (to be able to include such character sets as Chinese, Japanese, music, mathematics, etc.)
4. User-defined data types
5. Module and procedure definition capabilities
6. Pointers
7. "The concept of language evolution"

This last feature refers to the fact that the Fortran language is expected eventually to change in two directions—by the addition of new features such as those proposed in this document, and by removing some old features that are no longer useful. A number of FORTRAN features are declared to be "obsolete" in this document, which implies that they are targeted for removal in one of the succeeding Fortran revisions.

Fortran 90, however, has not removed any of the FORTRAN 77 older features. This is to maintain the "upward compatibility" that has always been a trademark of FORTRAN compilers. Programs written under an older standard could generally (with a very few exceptions such as CHARACTER data types) be run under new versions of the compiler. This is a desirable characteristic, since it has avoided the need for major overhauls of the huge battery of functional FORTRAN programs in existence. Thus, Fortran 90 will maintain upward compatibility, but FORTRAN users should be on the alert for the removal of some "archaic" language features by the time the *next* revision comes out (perhaps around the year 2000).



## OBSOLESCENT FEATURES

FORTRAN features in this category are those which are considered to have been redundant in FORTRAN 77, even though they are still used frequently. It was judged that better methods already exist for doing these jobs, and that programmers should adopt these better methods. If the use of some of these features becomes greatly decreased, it will be recommended that they be removed from future versions of Fortran. The features which have been designated obsolescent follow.

*Arithmetic IF* Other methods, such as logical and block IFs, are considered superior.

*Real and Double Precision DO Loop Variables* Though this was a feature added in FORTRAN 77 that had not been available before, the new Fortran 90 proposal recommends going back to the old restriction of only integer values controlling DO loops.

*Shared Terminal Statements for DO Loops* It is recommended that having one common statement as the terminal statement for several nested DO loops be disallowed. It is further recommended that the only terminal statements allowed for DO loops be END DO statements or CONTINUEs.

*Branching to an ENDIF statement from outside its IF block* It is recommended that this be made illegal. Branches can be made to the statement immediately following the ENDIF.

*Alternate RETURNS* It is suggested that this feature (which is discussed in Chapter 9) be replaced by setting a flag in the subroutine and then testing the flag, using IFs or the new CASE construct upon returning from the CALL.

*PAUSE Statement* This is a carryover from early times when operator intervention might be required in the middle of a long program—for instance, to mount a tape. This should not be necessary in today's environments.

*ASSIGN and ASSIGNED GO TO* This allowed the programmer to set a value of an integer variable using an ASSIGN statement and then execute a GO TO where the statement number that was the target of the GO TO was provided by the variable. Clearly, this made programs using such a device difficult to follow, and there are many other techniques that could be used to accomplish the ends that this statement pair dealt with.

*Assigned FORMAT Specifiers* Again, using the ASSIGN statement to provide FORMAT statement numbers to I/O statements. Clearly, this procedure makes program flow difficult to follow, and the goal can be better accomplished by using character variables to describe FORMATS, as is possible in FORTRAN 77.

*cH Edit Descriptor* This is a carryover of earlier means of declaring character constants and literal strings in formats by counting the characters in the string (n) and then preceding the string by an nH designator. Since it is much simpler and less error-prone to enclose literal strings in quotes, this is better.

## “Deprecated” Features (From an Earlier 8x Version of the Proposal)

An earlier version of a Fortran 8x proposal indicated that this set of FORTRAN features might be expected to become obsolescent when the new features of Fortran 90 go into general use. Programmers are encouraged to substitute the better methods Fortran 90 will provide. Features which fall into disuse may be recommended for deletion in later versions of the Standard. Thus, although this section was not carried forward into the last version of the revision, it does indicate a general displeasure with these features, which may well be operative when successive revisions of the language are proposed.

*Assumed-Size Dummy Arrays* The use of the asterisk (\*) to declare the size of dummy arrays in subprograms is discouraged. A feature of Fortran 90 called the assumed-shape array will take its place. An assumed-shape declaration might be of the form:

```
REAL X(:,), Y(1:)
```

*Passing an array element or Substring to a Dummy Array* This was not a feature generally encouraged even in earlier FORTRAN. It would now be replaceable by the capability in Fortran 90 to isolate a section of an array to be dealt with.

*BLOCK DATA Subprogram* This would no longer be necessary after the introduction of Fortran 90 modules.

*COMMON* The global data previously provided by COMMON areas would now be taken over by the use of modules.

*ENTRY Statements* It is claimed that the ENTRY statement was generally used to share data among sets of operations, and that the module facility will provide this capability. A module will now be able to declare data shared by several procedures, and these procedures may be a mixture of subroutines and functions, a capability not available in FORTRAN 77. However, we still think the ENTRY allows capabilities for using common code.

*EQUIVALENCE* Reuse of storage space, if necessary, will be provided by the ALLOCATE provision to dynamically allocate arrays.

*Statement Functions* These will be replaced by the more flexible Internal Function, which will allow its own specifications for data types and structures, available modules, and the like.

*Computed GO TO* It is considered that the new SELECT CASE construct will be superior.

The designers of Fortran 90 have indicated that they would like to see the language referred to in the future as Fortran, not FORTRAN.

## ARRAY OPERATIONS

Arrays of many dimensions, each dimension having a specified lower and upper bound, may be declared, as in FORTRAN 77. Further, array *segments* or *sections* may be pinpointed, by using subscript triplets in the subscript position for a dimension. A subscript triplet declares a beginning and ending subscript value to establish a range, and then a "stride," or step size, from initial to final value (analogous to a DO loop).

name (lower:upper:stride, . . .)

If an array C has been dimensioned to 20 locations, we can create two sections of C as follows:

C(2:12:3)      C(20:1:-3)

where the first is a four-element subset of C, including C(2), C(5), C(8), and C(11), and the second is a seven-element subarray consisting of C(20), C(17), C(14), C(11), C(8), C(5), and C(2), in that order.

Entire arrays can be processed by using a single reference to the array name. For example, array A of 100 entries can be initialized to 0 by the following statement:

```
REAL A(100)
A = 0
```

and whole arrays may be combined at once. For example, we may write, for arrays A, B, C, and D of the same length:

```
REAL A(200), B(200), C(200), D(200)
D = A*B + SQRT(C)
```

Further, operations may be performed on elements of an array meeting a certain condition, by using a WHERE statement:

```
REAL A(300)
WHERE (A.GT.90)
  A = A - 10.
ELSEWHERE
  A = A + 12.
END WHERE
```

Not only may arrays be segmented easily in Fortran 90, there are also a number of new intrinsic procedures that will operate on whole arrays or specified segments to perform specific array manipulations. These include:

**DOTPRODUCT (A,B)** which will take the dot product of two vectors (one-dimensional arrays) that must be of the same size

**MATMUL (A,B)** which will multiply together the matrix A (of size  $m \times k$ ) and the matrix B (of size  $k \times n$ ) to give a resulting matrix of size  $m \times n$ , according to the rules of matrix multiplication, which we discussed in the text

**SUM (A, DIM, MASK)** which will give the sum of all the elements of array A along dimension DIM and with the "mask" specified by the logical MASK.

DIM and MASK are optional. If DIM is not specified, it will sum all of the elements of the array A; otherwise it sums across the dimension position specified. If MASK is not specified, it will sum all elements (along dimension DIM). If MASK is specified, it will sum all elements (along dimension DIM) which satisfy the condition specified by MASK (e.g., A .LE. 30)

**MAXVAL (A,DIM,MASK)** and **MINVAL (A,DIM,MASK)** will return the maximum (or minimum) value of array A [along dimension DIM] for all values which satisfy MASK, if specified

**PRODUCT (A,DIM,MASK)** computes the product of all elements in array A [along dimension DIM] [which satisfy the condition MASK]

**TRANSPOSE (A)** takes the transpose of a two-dimensional array (matrix) A

**MAXLOC (A,MASK)** and **MINLOC (A,MASK)** return the location of the maximum (or minimum) value of array A [satisfying MASK]

**ALL (MASK, DIM)** returns a logical value indicating whether all of the values are true in MASK along dimension DIM

**ANY (MASK, DIM)** returns a logical value indicating whether any value is true in MASK along dimension DIM

**COUNT (MASK, DIM)** returns the number of true instances of MASK along dimension DIM

There are additional intrinsic procedures to inquire into the allocation, bound, size, or shape status of an array; to PACK or UNPACK an array using a MASK; to shift values in an array in a circular or end-off manner (using CSHIFT or EOSHIFT); and so forth. All of these save the programmer from having to write such procedures, which are quite useful in many applications programs.



## IMPROVED NUMERIC COMPUTATION CAPABILITIES

Fortran 90 will provide capabilities for specifying the desired precision and exponent range for a real value. This will greatly enhance number-crunching

facility, which is generally hampered by word size restrictions, as we have discussed.



## USER-DEFINED DATA TYPES

The programmer may create new data types out of the existing types available in Fortran (integer, real, double precision, complex, character, logical), available data structures such as arrays, and previously defined user data types. The derived-type definition must begin with the keyword TYPE followed by the name for the type being defined. The lines that follow provide a *template* for the derived type, the nature and structure of the components of this data type. The components may be arrays, or previously defined user data types. If the type COMPLEX had not already been defined for FORTRAN, it could be defined as:

```
TYPE COMPLEX
    REAL X,Y
END TYPE COMPLEX
```

and then when you wanted to declare variables Z and V as complex:

```
TYPE (COMPLEX) Z, V
```

Similarly, if we wanted to define a “player” in a dating game, with certain component characteristics:

```
TYPE PLAYER
    CHARACTER SEX
    CHARACTER*20 NAME
    INTEGER AGE
    INTEGER HEIGHT
    INTEGER WEIGHT
    INTEGER INCOME
END TYPE PLAYER
```

and we then wanted to compute the compatibility of a date pair:

```
TYPE DATEPAIR
    TYPE(PLAYER) GIRL, BOY
END TYPE DATEPAIR
```

This should greatly increase our capability to construct and manipulate various data types and structures.

A *component* of a complex defined data structure can then be identified using the component-selector operator %. For example,

```

TYPE (PLAYER) FRED
FRED%AGE = 21
FRED%NAME = 'FREDERICK THE GREAT'

```

and so on.

## USER-DEFINED OPERATIONS

The user can define unary or binary operations, using a Function subprogram to do so. Any user-defined unary operations will have precedence higher than all intrinsic operators, and any user-defined binary operations will have the lowest precedence. A defined operation must be explicitly declared in an INTERFACE statement; for example, to create a logical operator .XOR. :

```

INTERFACE OPERATOR (.OP.)
  FUNCTION XOR (X, Y)
    LOGICAL XOR, X, Y
    XOR = X.OR.Y .AND. (.NOT. (X.AND.Y))
    RETURN
  END
END INTERFACE

```

## MODULES

A module is a program unit that may contain data specifications and definitions that will then be accessible by another program unit. Since variables, data structures, and their values will thus be made equally accessible to any program unit, the need for constructs such as COMMON and BLOCK DATA will disappear.

In Fortran 90, a module may be used to declare types and dimensions of variables, may contain module subprograms, and may be used to define new data types (see Chapter 13). The contents of a module may be accessed by any program unit, through an appropriate USE statement. A module definition has the form:

```

MODULE name
  [module specification statements (type, dimension)]
  [module subprogram definitions]
END MODULE [name]

```

and its contents may be included in any program unit by a

```
USE name
```

statement. The module name is global, and thus may not be the same as any other global variable or program unit name. If any variable is declared PRIVATE in the module declaration area, it is not available outside of the module. This allows for *data encapsulation*—the definition of data is accomplished in one place—and *data hiding*—the ability to keep information which is not necessary for operation secure from programs that use the module.

## POINTERS

Pointers allow dynamic control of array size, and the construction of such data types as stacks, queues, trees, and graphs. A pointer name may refer to different storage space at different times in the program. A pointer may be associated with a target by a statement of the form:

pointer => target

where the target must be of the same type and rank as the pointer. The pointer variable must have been declared to have the attribute POINTER, and the target variable must have an attribute of either TARGET or POINTER. Such pointers may be used in constructing linked lists or as pointers in stacks or queues, as we discussed (and were handled in a more primitive fashion) in Chapter 13.

## RECURSION

Recursion will be included in Fortran 90. This means that a function or a subroutine may call itself. A function, for example, need only be declared RECURSIVE and specify a RESULT:

RECURSIVE FUNCTION NFACT(N) RESULT(FACTORIAL)

We refer the interested reader to our discussion of recursion in Chapter 9.

## OTHER INTERESTING CHANGES

### Free-form Statements and Comments

The new Fortran 90 will encourage the availability of free-form statement lines in Fortran, such that no columns are reserved for specific purposes. If the old, fixed-form line layout is opted for, or the only one available, it will have

the usual familiar restrictions: columns 1–5 for statement numbers, column 6 for continuation character, a comment indicated by a C or an \* in column 1. If free-form is available, lines may be up to 132 characters long, comments will be initiated on any line with a !, and the rest of that line will be taken as a comment. Continuation will be indicated by a & (ampersand) as the last nonblank character of a line to be continued. A statement separator will be the semicolon (;), allowing several short instructions to be written on one line, separated by semicolons.

## Variable Names and Type Declarations

The characteristics of several variables may be declared at once as, for example:

```
INTEGER, DIMENSION (100):: A, CAT, DOG
```

Variable names may be up to 31 characters long, beginning with a letter, and made up of letters, digits, or underscores (\_). This is quite a change from the 6-character restriction that was in force (due to the IBM 704) for over 30 years.

Furthermore, a defined array may be initialized in the declaration statement; for example,

```
INTEGER, DIMENSION (5) :: X = (/ 1, 2, 3, 4, 5 /)
```

## Extension of Logical Relation Symbols Available

The six logical relations on the system (.LT., .LE., etc.) may now also be written as <, <=, >, >=, ==, and /=.

## Character String Designation

Character strings may be preceded by an nH delimiter (though this is considered obsolescent), but much preferably they will be enclosed in single quotes ('') or double quotes ("").

## Additional Format Descriptors

Fortran 90 will allow format descriptors for octal (O), binary (B), and hexadecimal (Z) values. It also extends the format descriptors for real values to include engineering and scientific notation. An EN format descriptor (used like an F descriptor) describes a real value expressed with an exponent divisible by 3 and a multiplier in the range 1 to 1000. An ES (“scientific”) format descriptor creates an output value whose multiplier is in the range from 1 to 10.

## IMPLICIT NONE

An IMPLICIT NONE statement may be used to undo all built-in default typing. If it is used, no other IMPLICIT typing may be used in the same program.

## DO Loop Extensions

A DO loop may be terminated by an END DO statement, thus eliminating the need for a statement number on the terminal statement. For example, we may have:

```
DO I = 1, 100
    {loop body}
END DO
```

If there are nested loops, the END DO is simply matched up with its closest DO, just as in IF THEN/ENDIF blocks.

The CYCLE statement may be used in the range of a DO loop to force execution of the next *cycle* of the loop; that is, it has the effect of a branch to the CONTINUE or END DO statement, if such exists in the loop. It decreases the iteration count of the loop by one, alters the loop control variable by the step size, and repeats the range of the loop (if it has not terminated).

The EXIT statement may be included in the range of a DO loop, and has the effect of prematurely terminating the execution of the loop; control is passed to the first executable statement following the terminal statement of the loop.

## WHILE Loops

The DO WHILE ( ) statement (similar to that on the VAX we have discussed) will be implemented in Fortran 90. This will allow the construction of loops that repeat as long as a certain condition holds; they are *pre-test* loops of the form:

DO # WHILE (logical condition)	{or}	DO WHILE (log. cond.)
...		...
# CONTINUE		END DO

## SELECT CASE Construct

The SELECT CASE construct, which we discussed in the first section of Chapter 10, will be implemented in Fortran 90 to allow easy selection among alternatives to be performed. A feature found in several languages today is the “case”

structure, in which different segments of code may be executed if a particular expression matches the value or range of values specified for that case. Fortran 90 will implement a SELECT CASE feature of the form:

```

SELECT CASE (case-expression)
CASE case-selector1
    action1
CASE case-selector2
    action2
...
[ CASE (DEFAULT) ]
END SELECT

```

## Random Number Generator

A random number generator will be incorporated into every Fortran 90 standard compiler. It will be of the form of a subroutine, and a call to it will be of the form:

```
CALL RANDOM_NUMBER (X)
```

where X is a real variable to contain the resulting random value in the range from 0 to 1. We would have preferred to see it as a function, similar to the one we have used in this book. The random generator may be seeded (see our discussion in Chapter 13) by using the subroutine RANDOM\_SEED. A simple use of this subroutine would be as follows:

```
CALL RANDOM_SEED (PUT = SEED)
```

## NAMELIST Feature

The NAMELIST I/O feature will be included in Fortran 90. We refer the interested reader to our extended discussion of NAMELIST in Chapter 8, exercise 12.

## INCLUDE Statement

The source code of a program may have a line of the form:

```
INCLUDE character-constant
```

where the character string may name a text file whose contents is then incorporated into the program source code at the point where the INCLUDE statement appeared.

## INTENT Declarations for Subprogram Variables

The INTENT attribute of a dummy variable in a subprogram may be declared to be IN (that is, the argument may not be redefined in the subprogram), OUT (the argument *must* be defined in the subprogram, and the actual argument it is associated with on the call must be definable—that is, a variable), or INOUT (the argument may both be referenced and defined in the procedure). A dummy argument whose INTENT is not declared is subject to any limitations of the actual argument associated with it.

```
INTENT (IN) A
INTENT (OUT) :: MAX, RANGE
```

## Other Useful Procedures Available

A variety of useful inquiry functions will be available in Fortran 90, such as DIGITS (X), which will indicate the number of significant digits available; EPSILON (X), which will indicate the value that is almost negligible compared to 1 on the system (or one can use TINY (X), which returns the smallest real that can be stored); HUGE (X), which will return the largest number available on the system; MAXEXPONENT (X) and MINEXPONENT (X); and so on.

A number of bit manipulation functions will also be provided in Fortran 90, which will test a specific bit (BTEST), set a bit (IBSET), perform logical ANDs or ORs, extract a bit (IBITS), complement a value, or perform a logical shift (ISHFT) or a circular shift (ISHFTC).

## Other Intrinsic Subroutines

We already introduced the new random number subroutines in Fortran 90. In addition, the following subroutines are new:

DATE\_AND\_TIME (DATE, TIME, ZONE, VALUES)—which can be used to obtain the date and time

MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)—which copies bits from one integer location into another

SYSTEM\_CLOCK (COUNT, COUNT\_RATE, COUNT\_MAX)—which will obtain information from the system clock

There are other new functions available, but we have described the ones most likely to be useful to us in our problem solving as scientists and engineers.

All in all, it is clear that the dedicated programmer will have an array of interesting and powerful new tools in the Fortran 90 revision when it becomes available.

# ANSWERS TO SELECTED EXERCISES



## ◆ INTRODUCTION

**4.** The EDVAC incorporated von Neumann's "stored-program" concept, and so could change programs as readily as it changed the data programs were to operate on.

**8.** A 25 in binary is  $11001_2$ . Thus, in a 16-bit word, a +25 would be  $0000000000011001_2$ , and so its one's complement, representing -25, would be  $11111111100110_2$ .

**9.** A 36 in base 10 is  $100100_2$ . In a 16-bit word, a positive 36 would be represented as  $0000000000100100$ ; thus its one's complement form would be  $11111111011011$ , and adding 1 to that to get the *two's complement* would give the result  $11111111011100$ .

**12.** The *largest* value that could be stored would have all binary 1's in both the multiplier and in the exponent, with a positive sign of the number and the exponent, and so would be:

$$0\ 0\ 111111\ 1111111_2 \quad \text{or} \quad 00111111111111_2$$

which means that the power of 2 would be  $111111_2$ , or  $63_{10}$ , and the multiplier would be  $.1111111_2$ . Thus the value itself is

$$0.1111111_2 \times 2^{63}, \text{ or } 1111111_2 \times 2^{55}$$

which is  $255_{10} \times 2^{55}$ , or  $255 \times 3.603 \times 10^{16}$ , or roughly  $9.187 \times 10^{18}$  (that is, 9187 followed by 15 zeros).

The *smallest* (non-zero) value that could be stored would still have to have a 1 as the most significant digit of the multiplier, so the multiplier would be  $(.)\ 10000000$ , and the power of two should be the largest possible *negative* exponent, that is,  $-111111_2$ . So the value would be stored as:

$$0\ 1\ 111111\ 10000000_2, \quad \text{or} \quad 011111110000000_2$$

and would be equal to  $0.1_2 \times 2^{-63}$ , or  $0.5_{10} \times 2^{-63}$ , which is roughly  $0.5 \times 10^{-19}$ , or  $5.421 \times 10^{-20}$ . Note: these numeric equivalences were determined using a short

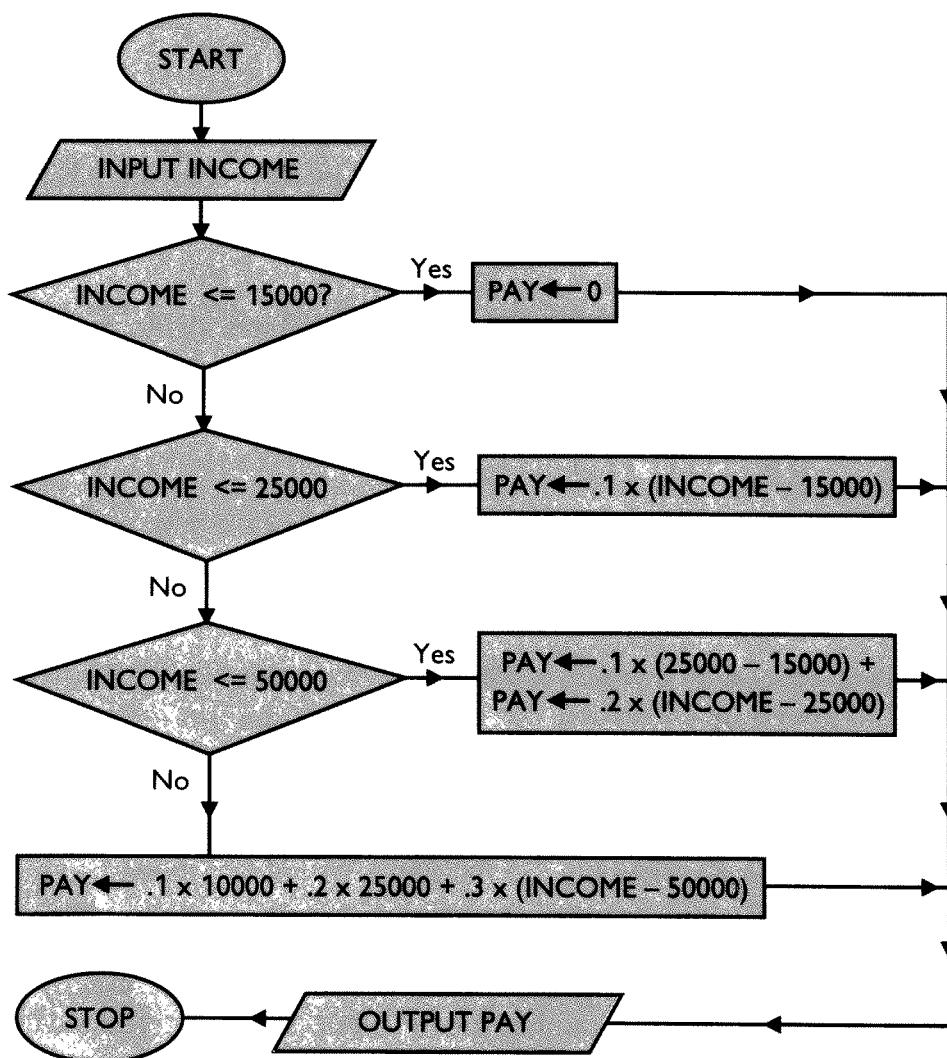
program. Note also that attempting to compute values larger than this largest value on your 16-bit machine would cause an *overflow*, and values less than the smallest value (but not 0) would cause *underflow*.

15. MUL      A, B, Y  
       DIV      C, D, Z  
       SUB      Y, Z, Y  
       ADD      Y, X, Y

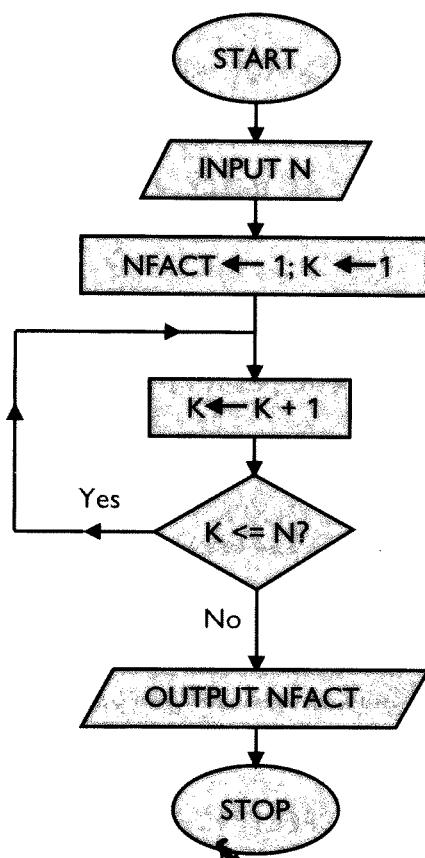
(We have assumed that SUB R, S, T subtracts S from R and stores the result in T. We also had to create a temporary extra location Z to hold the result of the division of C by D before it could be combined with the contents of Y by subtraction.)

## CHAPTER I

1f.



3.



## CHAPTER 2

2. line 3.

$$J = C+B/K+N = 4.0+6.0/2+3 = 4.0+3.0+3 = 10.0 \text{ (stored as 10)}$$

$$D = -A+B = -3.5 + 6.0 = 2.5$$

$$J = N*N**K = 3*3**2 = 3*9 = 27$$

3.  $X = (A^{**2} + B^{**2})^{**}(1.0/3.0)$   
 $X = 1.0/2.0*A*Y^{**2}$

7. PROGRAM ROXANNE

```

* CALCULATE HEIGHT OF STREAM OF WATER FROM STEVE'S HOSE *
REAL A, V, VX, VY, X, T, G
PARAMETER ( G = 32.2)
PRINT*, 'INPUT ANGLE OF HOSE (IN RADIANS) AND VELOCITY'
READ*, A, V
PRINT*, 'INPUT DISTANCE OF HOSE FROM BUILDING (IN FEET)'
READ*, X
  
```

614 ANSWERS TO SELECTED EXERCISES

```
VX = V*COS(A)
VY = V*SIN(A)
T = X/VX
Y = VY*T - 0.5*G*T*T
PRINT*, 'THE HEIGHT AT WHICH THE WATER HITS THE BUILDING'
PRINT*, 'IS ', Y, ' FEET.'
STOP
END
```

11. (Generalized program, second part of problem)

```
PROGRAM HEAT
CHARACTER NAME*8
REAL SPHEAT, AMOUNT, TEMPA, TEMPB, CALS
*****
* PROBLEM TO CALCULATE NUMBER OF CALORIES REQUIRED TO RAISE *
*
* A SPECIFIED AMOUNT OF SOME ELEMENT FROM A TO B DEGREES C. *
*****
PRINT*, 'INPUT THE NAME OF THE ELEMENT IN SINGLE QUOTES'
READ*, NAME
PRINT*, 'INPUT THE SPECIFIC HEAT AND AMOUNT OF ELEMENT'
READ*, SPHEAT, AMOUNT
PRINT*, 'INPUT THE LOW AND HIGH TEMPERATURES, IN CELSIUS'
READ*, TEMPA, TEMPB
CALS = SPHEAT*AMOUNT*(TEMPB-tempa)
PRINT*, 'IT WILL TAKE ', CALS, ' CALORIES TO RAISE ',
PRINT*, AMOUNT, ' GRAMS OF ', NAME, ' FROM ', TEMPA,
PRINT*, ' DEGREES TO ', TEMPB, ' DEGREES CELSIUS'
STOP
END
```

15. PROGRAM FORCE

```
REAL G, POUNDM, WEIGHT, EARTH, RADIUS
***** WHAT FORCE DOES THE EARTH EXERT ON YOU? *****
PARAMETER (G = 6.67 E-11, POUNDM = 0.453592, EARTH =
& 5.98 E 24, RADIUS = 6378000.0)
PRINT*, 'INPUT YOUR WEIGHT (IN POUNDS)'
READ*, WEIGHT
FORCE = G*EARTH*WEIGHT*POUNDM/(RADIUS*RADIUS)
PRINT*, 'THE FORCE ACTING ON YOU AT THE EQUATOR WOULD BE ',
PRINT*, FORCE, ' NEWTONS'
STOP
END
```

**19.** REAL MICRO

```
MICRO = 1.0 E-6* 100*365.25*24*60
PRINT*, 'VON NEUMANN''S OPTIMUM LECTURE LENGTH IS ',
PRINT*, MICRO, ' MINUTES'
```

(This should come out to be about 52.596 minutes.)

**23.**

```
*      FIND THE SMALLER AND LARGER OF TWO (REAL) VALUES      *
REAL A, B, SMALL, BIG
READ*, A, B
SMALL = (A + B - ABS(A-B))/2.0
BIG = A + B - SMALL
```

Notice that this technique would also work on integer values.



## CHAPTER 3

**1.** INTEGER YEAR

```
READ*, YEAR
IF (YEAR - YEAR/4*4 .EQ. 0) THEN
    PRINT*, YEAR, ' IS LEAP'
ELSE
    PRINT*, YEAR, ' IS NOT LEAP'
ENDIF
```

**5.** REAL WEIGHT, TOTAL

```
K = 0
TOTAL = 0
READ*, WEIGHT
5 IF (WEIGHT .GE. 0) THEN
    K = K + 1
    TOTAL = TOTAL + WEIGHT
    READ*, WEIGHT
    GO TO 5
ENDIF
PRINT*, 'THE AVERAGE WEIGHT IS ', TOTAL/K
```

This has been implemented with a While-type structure. It could also have been done with a Repeat/Until structure. Try it.

**9.** Set up a *table* of the values in the loops and how they change to check out these program sequences. For example,

1)	A	N
	2.0	1
	4.0	2
	16.0	3
	256.0	4

The value of A printed out will be 256.0

- 2) The value of J printed out is 4
- 3) The value of A printed out is 9.0

10. 2)	M	J
	1	0
	3	1
	9	4
	27	13
	81	40

The value of J printed out is 40

**15.** REAL SUM  
 SUM = 0.0  
 N = 200  
 3        SUM = SUM + 1.0/N  
 N = N - 1  
 IF (N .GE. 101) GO TO 3  
 PRINT\*, 'THE SUM OF THESE FRACTIONS IS ', SUM

**18.** PROGRAM TAX  
 REAL INCOME, TAX  
 PRINT\*, 'INPUT AMOUNT OF TAXABLE INCOME'  
 READ\*, INCOME  
 IF (INCOME .LE. 5000.0) THEN  
     PRINT\*, 'NO TAX ON \$', INCOME  
     STOP  
 ELSEIF (INCOME .LE. 20000.0) THEN  
     TAX = .12\*(INCOME - 5000.0)  
 ELSEIF (INCOME .LE. 45000.0) THEN  
     TAX = .20\*(INCOME - 20000.0) + .12\*(20000. - 5000.)  
 ELSE  
     TAX = .28\*(INCOME - 45000.0) + .2\*25000. + .12\*15000.

```

ENDIF
PRINT*, 'THE TAX ON $',INCOME, ' IS $', TAX
STOP
END

20. PROGRAM DULONG
CHARACTER ELEMNT*8
REAL SPHEAT, ATOMWT
PRINT*, ' ELEMENT DULONG-PETIT PRODUCT'
PRINT*, ' _____
N = 1
6      PRINT*, 'INPUT ELEMENT NAME (ENCLOSED IN QUOTES),'
      PRINT*, 'SPECIFIC HEAT, AND ATOMIC WEIGHT'
      READ*, ELEMNT, SPHEAT, ATOMWT
      PRINT*
      PRINT*, ELEMNT, '      ', SPHEAT * ATOMWT
N = N + 1
IF (N .LE. 6) GO TO 6
STOP
END

```

The PRINT\* instruction with no list will print a blank line between successive entries in the table, making it more readable.

## CHAPTER 4

4. (1) Doubly nested loops ending on the same statement are legal as long as no branch to the terminal statement is taken from a loop other than the innermost loop, and if there is no "work" to be done between the end of the inner loop and that of the outer loop. However, many programmers frown on this practice and insist that each loop have its own unique terminal statement.

(2) This loop is *legal*, that is, it will not create any compiler errors. However, in FORTRAN 77 it will be recognized as an "impossible" loop, and be skipped entirely. (*Note:* in earlier versions of FORTRAN, this loop would be executed *once*, and would print out the value 5.)

(3) This is actually illegal, since the arithmetic IF will attempt to jump *into the range* of the DO loop for negative or zero values of J. However, some compilers do not pick this out. For example, a VAX/VMS compiler allowed the following code:

## 618 ANSWERS TO SELECTED EXERCISES

```
K = 3
IF (K - 3) 10, 20, 10
10 DO 30 I = 1, 3
      PRINT*, I
20     PRINT*, 'JUMP'
30 CONTINUE
```

and the resulting printout was:

```
JUMP
      1
JUMP
      2
JUMP
      3
JUMP
```

**5.** (2) The value of K printed out will be 15, that is, the sum of the integers from 1 through 5. It does not matter that the sum is performed four times by the outer (J) loop, since the K accumulator is reinitialized to 0 every time. Question: What would be printed out if the K = 0 statement were placed *before* the outer (J) loop?

**8.** Simply insert, in the Repeat/Until loop which begins with statement 55, and immediately after BOUNCE = BOUNCE\*R, the line:

```
R = R * .98
```

**12.** PROGRAM JIMMY
REAL MONEY
INTEGER YEARS
YEARS = 0
MONEY = 10.0
50 IF (MONEY .LE. 1000000.0) THEN
 MONEY = MONEY\*2
 YEARS = YEARS + 2
 GO TO 50
ENDIF
PRINT\*, 'IT TOOK JIMMY ', YEARS, ' YEARS TO BECOME A'
PRINT\*, 'MILLIONAIRE'
STOP
END

```

15. PROGRAM BATSON
      INTEGER DAYS, ORDERS, TOTAL
      DAYS = 1
      ORDERS = 24
      TOTAL = 1000 - ORDERS
***** REPEAT UNTIL PRODUCTION + BACKUP IS EXHAUSTED *****
60  CONTINUE
      DAYS = DAYS + 1
      ORDERS = ORDERS*3
      TOTAL = TOTAL + 1000 - ORDERS
      IF (TOTAL .GT. 0) GO TO 60
      PRINT*, 'IT TOOK BILLY ', DAYS, ' DAYS TO CATCH UP'
      PRINT*, 'WITH ;', ORDERS, ' ORDERS, AND A TOTAL'
      PRINT*, 'PRODUCTION OF ', DAYS*1000
      STOP
      END

19. NRAB = 0
      NFOX = 0
      PRINT*, 'YEAR', 'RABBITS', 'FOXES'
      DO 20 I = 1, 10
          NRAB = NRAB*1.05
          NFOX = NFOX*1.02
          PRINT*, I, NRAB, NFOX
20  CONTINUE

```

To modify as indicated, change the line after the DO 20 to:

$$\text{NRAB} = \text{NRAB} * 1.05 - .07 * \text{NFOX}$$

```

22. PROGRAM NUMDIG
      INTEGER NUM, N, NDIG
      N = NUM
      NDIG = 0
10  IF (N .GT. 0) THEN
      N = N/10
      NDIG = NDIG + 1
      GO TO 10
    ENDIF
    PRINT*, NUM, ' HAS ', NDIG, ' SIGNIFICANT DIGITS.'
    STOP
    END

```


**CHAPTER 5**

2. LOGICAL L

REAL X

READ\*, X

L = X .GT. 5.0

5. CHARACTER WORD\*8

READ\*, WORD

LEN = 8

DO 50 L = 1, 4

PRINT\*, 'THE SUBSTRINGS OF ', WORD, ' OF LENGTH ', L

\*\*\*\*\* RANGE STEP SIZE IS K \*\*\*\*\*

K = L - 1

\*\*\*\*\* N IS LAST BEGINNING POSITION FOR SUBSTRING \*\*\*\*\*

N = LEN - K

DO 40 J = 1, N

PRINT\*, WORD (J : J+K)

40 CONTINUE

50 CONTINUE

STOP

END

10. PROGRAM LEIBNZ

REAL PI, TERM

DOUBLE PRECISION DPI, DPTERM

MULT = 1

PI = 0.0

TERM = 1.0

DPI = 0.0 D 0

DPTERM = 1.0 D 0

DO 80 I = 1, 10

DO 60 J = 1, 100

PI = PI + TERM\*MULT

DPI = DPI + DPTERM\*MULT

TERM = TERM + 2.0

DPTERM = DPTERM + 2.0 D 0

MULT = MULT\*(-1)

60 CONTINUE

PRINT\*

PRINT\*, 'AFTER ', I\*100, 'TERMS, THE SUMS ARE:'

PRINT\*, 'SINGLE PRECISION : ', PI\*4.0

PRINT\*, 'DOUBLE PRECISION : ', DPI\*4.0 D 0

```

80 CONTINUE
STOP
END

17. PROGRAM HISTO
CHARACTER A*120
IMPLICIT INTEGER (K)
REAL GRAMS
DATA K1,K2,K3,K4,K5,K6,K7,K8,K9,K10 / 10*0 /
DO 20 I = 1, 800
    READ*, GRAMS
    IF (GRAMS .LE. 110.0) THEN
        K1 = K1 + 1
    ELSEIF (GRAMS .LE. 120.0) THEN
        K2 = K2 + 1
    ...
    ENDIF
20 CONTINUE
A =
DO 30 I = 1, K1
    A(I:I) = '*'
30 CONTINUE
PRINT*, A
A =
DO 40 I = 1, K2
    ...
    { etc. }
40 CONTINUE
STOP
END

```



## CHAPTER 6

```

3. PROGRAM CHESS
CHARACTER ACROSS*41, DOWN*41
DO 20 I = 1, 41
    ACROSS(I:I) = '_'
    DOWN(I:I) = '|'
20 CONTINUE
DO 30 I = 1, 41, 5
    DOWN(I:I) = '|'
30 CONTINUE
PRINT 33, ACROSS
33 FORMAT(' ', 20X, 'CHESSBOARD'//5X,A41)

```

## 622 ANSWERS TO SELECTED EXERCISES

```
DO 50 J = 1, 8
    DO 40 K = 1, 4
        PRINT 44, DOWN
44    FORMAT(5X,A41)
40    CONTINUE
        PRINT 44, ACROSS
50    CONTINUE
END
```

This program took advantage of the information that, on the printer, there are roughly 10 columns per inch and 8 rows per inch.

**8. PROGRAM KASH**

```
CHARACTER NAME*20
INTEGER TWO, SIX, ANS
OPEN (3, FILE = 'CASH', STATUS = 'OLD')
TWO = 0
SIX = 0
PRINT 22
22 FORMAT('1', 40X, 'NAMES OF THOSE WHO ANSWERED ''8'' ')
DO 200 I = 1, 1000
    READ (3, 44) ANS, NAME
44    FORMAT(/ 44X, I1// 60X, A20)
        IF (ANS .EQ. 8) THEN
            PRINT 55, NAME
55    FORMAT(20X, A20)
        ELSEIF (ANS .EQ. 2) THEN
            TWO = TWO + 1
        ELSEIF (ANS .EQ. 6) THEN
            SIX = SIX + 1
        ENDIF
200    CONTINUE
        PRINT 66, TWO, SIX
66    FORMAT(///10X, I4, ' ANSWERED ''2'' AND ', I4,
& ' ANSWERED ''6'' TO THE QUESTION')
        CLOSE 3
        STOP
    END
```

**11. PROGRAM LOWER**

```
CHARACTER LINE*80
INDEXA = ICHAR('A')
INDEXZ = ICHAR('Z')
LOW = ICHAR('A')
```

```

* NDIF IS THE CONVERSION FACTOR FROM UPPER TO LOWER CASE *
    NDIF = LOW - INDEXA
***** ASSUME THERE ARE 100 LINES OF TEXT TO READ *****
    OPEN (4, FILE = 'STORY', STATUS = 'OLD')
    DO 100 I = 1, 100
        READ 33, LINE
33     FORMAT (A80)
        DO 50 J = 1, 80
            C = LINE(J:J)
            INDEXC = ICHAR(C)
            IF (INDEXC.GE.INDEXA .AND. INDEXC.LE.INDEXZ) THEN
* IF IN RANGE OF CAPITALS, CONVERT INDEX TO CORRESPONDING L.C. *
                NEW = INDEXC + NDIF
                C = CHAR(NEW)
                LINE(J:J) = C
            ENDIF
50     CONTINUE
        PRINT 55, LINE
55     FORMAT(' ', A80)
100    CONTINUE
        STOP
    END

```

- 15.** To print out the positive integers from 1 through 100, with exactly one blank between each, a "cheat" would be to print them all (except 100) with leading zeros:

```

        PRINT 22, (N, N = 1, 100)
22     FORMAT ('0', 2X, 10I4.3)

```

However, if you want to print them without the leading zeros, and still have a blank between, it requires a bit more work:

```

        PRINT 33, (N, N = 1, 100)
33     FORMAT('0', 2X, 9I2, I3/ 8('0',10I3/), '0', 9I3, I4)

```

## CHAPTER 7

- 6. PROGRAM INVERT**
- ```

REAL A(200), TEMPA
READ*, A
K = 200
DO 20 I = 1, 100

```

```

        TEMPA = A(I)
        A(I) = A(K)
        A(K) = TEMPA
        K = K - 1
20    CONTINUE
        PRINT*, A
        STOP
        END

7.   PROGRAM BASE
        INTEGER N, B, NQ, REM(20)
        PRINT*, 'ENTER NUMBER AND BASE TO CONVERT TO'
        READ*, N, B
        PRINT 9, N, B
9     FORMAT('0 THE NUMBER ', I6, ' IN BASE ', I2, ' IS:')
        I = 0
20    IF (N .NE. 0) THEN
            NQ = N/B
            I = I + 1
            REM(I) = N - NQ*B
            N = NQ
            GO TO 20
        ENDIF
        PRINT 11, (REM(J), J = I, 1, -1)
11    FORMAT('0', 4X, 20I1)
        STOP
        END

```

For example, given 66 and 3 as input values, this program prints out:

```
THE NUMBER      66 IN BASE  3 IS:
      2110
```

```

13.   REAL C (30, 40)
      LOGICAL SWITCH
      ...
5     SWITCH = .FALSE.
      DO 20  I = 1, 29
          IF ( C(I,1) .LT. C(I+1,1) ) THEN
              DO 15  J = 1, 40
                  TEMP = C(I,J)
                  C(I,J) = C(I+1,J)
                  C(I+1,J) = TEMP
15     CONTINUE

```

```

        SWITCH = .TRUE.
    ENDIF
20  CONTINUE
    IF (SWITCH) GO TO 5

17. PROGRAM PERFECT
    INTEGER FACTOR(100), NSUM
    FACTOR(1) = 1
    DO 100 N = 5, 500
        K = 1
        NSUM = 1
        DO 50 J = 2, N/2
            IF ( MOD(N,J) .EQ. 0) THEN
                K = K + 1
                FACTOR(K) = J
                NSUM = NSUM + J
            ENDIF
50      CONTINUE
            IF ( N. EQ. NSUM) PRINT 55, N, (FACTOR(I), I = 1,K)
55      FORMAT(//'0', I5, ' IS A PERFECT NUMBER, AND ITS ',
& 'FACTORS ARE://('0', 2X, 20I5) )
100   CONTINUE
    STOP
    END

20. PROGRAM CALEND
    CHARACTER*2 CAL(12, 6, 7), DAY(7)*3
    DATA DAY / 'SUN', 'MON', 'TUE', 'WED', 'THU','FRI','SAT'/


```

You will have a loop index (say, M) which goes from 1 to 12 for the months; each time you fill a calendar for a month, store it in "page" M (e.g., CAL(M, I, J) = . . .). Fill all 12 before you begin printing. Then, set up a print loop like the following:

```

L = 1
M = 4
DO 200 LOCK = 1, 3
PRINT 88, (NAME(K), K = L,M), ((DAY(J),J = 1,7), K = 1,4)
88 FORMAT(//11X,A10,3(22X,A10)//2X, 4(7A4,4X)//)
PRINT 99, (((CAL(K, I, J), J = 1, 7), K = L, M), I = 1,6)
99 FORMAT('0', 4 (7A4,4X))
L = L + 3
M = M + 3
200 CONTINUE

```

**22.** This program implements the *second* diagram.

```

PROGRAM DIAMOND
CHARACTER A(79)
PRINT*, 'INPUT THE (ODD) NUMBER OF ROWS FOR THE DIAMOND'
READ*, N
DO 30 I = 1, N, 2
    DO 15 J = 1, 79
        A(J) = ' '
15      K = (81 - I)/2
        DO 20 L = K, K + I - 1
20      A(L) = '+'
        PRINT 22, A
22      FORMAT(2X, 79A1)
        DO 40 I = N-2, 1, -2
            DO 25 J = 1, 79
25            A(J) = ' '
            K = (81 - I)/2
            DO 30 L = K, K + I - 1
30            A(L) = '+'
            PRINT 22, A
40      CONTINUE
      STOP
END

```

**24.** PROGRAM COMBO

```

INTEGER C(0:20, 0:20)
DO 100 N = 0, 20
    C(N,0) = 1
    C(N,N) = 1
    DO 80 M = N + 1, 20
        C(N, M) = 0
80      CONTINUE
    DO 90 M = 1, N-1
        C(N,M) = C(N-1,M-1) + C(N-1,M)
90      CONTINUE
100     CONTINUE
    DO 200 N = 0, 20
        PRINT 199, (C(N,M), M = 0, 20)
199     FORMAT('0', 21I6)
200     CONTINUE
END

```

```

26. PROGRAM RELAX
REAL V(8,8)
CHARACTER ANS*3
DO 10 I = 2, 7
    DO 10 J = 1, 8
        V(I,J) = 0.
        V(1,J) = 100.
        V(8,J) = 40.
10 CONTINUE
***** ITERATION *****
20 CONTINUE
    PRINT*
    PRINT*
    DO 30 I = 2, 7
        DO 30 J = 2, 7
            V(I,J) = ( V(I,J-1) + V(I,J+1) + V(I-1,J)
&                                + V(I+1,J) )/4.0
30 CONTINUE
    DO 40 I = 1, 8
        PRINT 33, (V(I,J), J = 1, 8)
33     FORMAT(2X, 8F6.2)
40 CONTINUE
    PRINT*, 'DO YOU WANT ANOTHER ITERATION? ''YES''?'
    READ*, ANS
    IF (ANS .EQ. 'YES') GO TO 20
    STOP
END

```



## CHAPTER 8

This chapter mostly has "try-it" exercises, for you to test out various error conditions and limitations on your machine, so solutions to most of these exercises are not appropriate. We have included solutions to a few of the problems.

```

13. PROGRAM OVER
N = 1
DO 20 I = 1, 100
    N = N*2
    PRINT*, I, N
20 CONTINUE

```

This program cuts out after printing values up through I of 30, N of

1073741824; on the next iteration of the loop, it prints out an error message about an “integer trap, arithmetic overflow”. A second program, beginning at N of 1073741824 + 1073741800, and stepping the value by 1, reaches an arithmetic overflow after the value of 2147483647, which is the largest integer on our machine.

A different program was run to test for underflow:

```

A = 1.0
DO 20 I = 1, 500
    A = A/2.0
    PRINT*, I, A
20 CONTINUE

```

This program did not terminate, but the values went to zero after the last meaningful value printed of 2.9387359 E-39 (I of 128). Thus our machine reports no underflow condition; the values just go to zero. Your machine may behave differently.

### 18. PROGRAM RATION

```

PRINT*, 'ENTER FOUR INTEGERS REPRESENTING TWO RATIONALS'
READ*, I, J, K, L
LOW = J*L
***** ADDITION *****
NUM = I*L + K*j
PRINT 5, I, J, '+', K, L, NUM, LOW
5 FORMAT('0', 5X, I3,'/',I3,A2,I3,'/',I3, ' = ',I4,'/',I4)
NUM = I*L - K*j
PRINT 5, I, J, '-', K, L, NUM, LOW
NUM = I*K
PRINT 5, I, J, '*', K, L, NUM, LOW
***** DIVISION *****
NUM = I*L
LOW = J*K
PRINT 5, I, J, '/', K, L, NUM, LOW
END

```

*Note:* this program does not necessarily print out the results in their lowest (fractional) terms. If we had a function IGCD (to be discussed in Chapter 9) that would give the greatest common divisor (GCD) of two arguments, we could apply that function to NUM and LOW each time we calculated them, for example:

```
NDIV = IGCD(NUM, LOW)
```

and then we could divide both NUM and LOW by NDIV to put them in their lowest terms.



## CHAPTER 9

```

4. MODULO(M,N) = M - M/N*N
   READ*, IY
   IF (MODULO(IY,4) .EQ. 0) THEN
      IF (MODULO(IY,100) .EQ. 0) THEN
         IF (MODULO(IY,400) .EQ. 0) THEN
            PRINT 5, IY
         ELSE
            PRINT 6, IY
         ENDIF
      ELSE
         PRINT 5, IY
      ENDIF
   ELSE
      PRINT 6, IY
   ENDIF
5 FORMAT('0', I5, ' IS A LEAP YEAR')
6 FORMAT('0', I5, ' IS NOT A LEAP YEAR')

```

9. XMAX(V0,A) = V0\*V0\*SIN(A)/32.2

```

11. SUBROUTINE PALIN (STRING)
   CHARACTER STRING*(*)
   LOGICAL PAL
   L = LEN(STRING)
   LMAX = L/2
   PAL = .TRUE.
   I = 1
20  IF (I.LE.LMAX .AND. PAL) THEN
      IF ( STRING(I:I) .NE. STRING(L:L) ) PAL = .FALSE.
      I = I + 1
      L = L - 1
      GO TO 20
   ENDIF
   IF (PAL) THEN
      PRINT*, STRING, ' IS A PALINDROME'
   ELSE
      PRINT*, STRING, ' IS NOT A PALINDROME'
   ENDIF
   RETURN
END

```

## 630 ANSWERS TO SELECTED EXERCISES

17. SUBROUTINE CHESS (QUEENS)  
INTEGER QUEENS(5,2), BOARD(8,8)  
DO 10 I = 1, 8  
 DO 10 J = 1, 8  
 BOARD(I,J) = 0  
10 CONTINUE  
DO 30 I = 1, 5  
 M = QUEENS(I,1)  
 N = QUEENS(I,2)  
 BOARD(M,N) = 1  
\*\*\*\*\* ROWS AND COLUMNS COVERED \*\*\*\*\*  
DO 15 K = 1,8  
 BOARD(M,K) = 1  
 BOARD(K,N) = 1  
15 CONTINUE  
\*\*\*\*\* DIAGONALS COVERED \*\*\*\*\*  
J = M  
K1 = N  
K2 = N  
DO 20 L = 1, M - 1  
 J = J - 1  
 K1 = K1 - 1  
 IF (K1.GE.1) BOARD(J,K1) = 1  
 K2 = K2 + 1  
 IF (K2.LE.8) BOARD(J,K2) = 1  
20 CONTINUE  
J = M  
K1 = N  
K2 = N  
DO 25 L = M + 1, 8  
 J = J + 1  
 K1 = K1 - 1  
 IF (K1.GE.1) BOARD(J,K1) = 1  
 K2 = K2 + 1  
 IF (K2.LE.8) BOARD(J,K2) = 1  
25 CONTINUE  
30 CONTINUE  
\*\*\*\*\* IS ENTIRE BOARD COVERED? \*\*\*\*\*  
DO 40 I = 1, 8  
 DO 40 J = 1, 8  
 IF (BOARD(I,J).EQ.0) THEN  
 PRINT\*, 'BOARD IS NOT COVERED'  
 RETURN  
 ENDIF

```

40  CONTINUE
    PRINT*, 'BOARD IS COVERED'
    RETURN
    END

21. FUNCTION LITTLE (MAT, M, N)
    INTEGER MAT(M,N)
    LITTLE = MAT(1,1)
    DO 20 I = 1, M
        DO 20 J = 1, N
            IF (MAT(I,J) .LT. LITTLE) LITTLE = MAT(I,J)
20  CONTINUE
    RETURN
    END

```

**25.** (assuming the compiler allows recursive subprograms)

```

FUNCTION NFACT(N)
IF (N .LE. 1) THEN
    NFACT = 1
    RETURN
ELSE
    NFACT = N*NFACT(N-1)
ENDIF
END

```

In Fortran 90, the entry line would have to be:

```
RECURSIVE FUNCTION NFACT(N) RESULT(NN)
```

and the recursive call line would have to read:

```
NN = N*NFACT(N-1)
```

```

33. FUNCTION SQROOT(A)
DATA EPSI/0.0001/
N = 1
B = 2.0
GO TO 20
ENTRY CUBRT(A)
N = 2
B = 3.0
20 X = A/B
30 CONTINUE

```

```

XP = (N*X + A/X**N)/B
DIFF = ABS(X/XP - 1.0)
X = XP
IF (DIFF .GE. EPSI) GO TO 30
SQROOT = XP
RETURN
END

```

*Note:* The function *should* equate all entry names for the subprogram, so that storing the result into SQROOT should have the effect of storing it into CUBRT if that entry had been taken. If it does not do so on your system (we have encountered such problems), then you can test on the value of N to see which name the result should be stored in; for example:

```

IF (N .EQ. 1) THEN
    SQROOT = XP
ELSE
    CUBRT = XP
ENDIF

```

35. CALL TICTAC (BOARD, \*10, \*20, \*30)  
 ...  
 SUBROUTINE TICTAC (BOARD, \*, \*, \*)  
 INTEGER BOARD(3,3), SUM(8)  
 SUM(7) = 0  
 SUM(8) = 0  
 DO 20 I = 1, 3  
 SUM(I) = 0  
 SUM(I+3) = 0  
 DO 10 J = 1, 3  
 \*\*\*\*\* ROW AND COLUMN SUMS \*\*\*\*\*  
 SUM(I) = SUM(I) + BOARD(I,J)  
 SUM(I+3) = SUM(I+3) + BOARD(J,I)  
 10 CONTINUE  
 \*\*\*\*\* DIAGONAL SUMS \*\*\*\*\*  
 SUM(7) = SUM(7) + BOARD(I,I)  
 SUM(8) = SUM(8) + BOARD(I, 4-I)  
 20 CONTINUE  
 DO 30 I = 1, 8  
 IF (SUM(I).EQ.0) RETURN 1  
 IF (SUM(I).EQ.3) RETURN 2  
 30 CONTINUE  
 DO 40 I = 1, 8  
 DO 40 J = 1, 8  
 IF (BOARD(I,J).EQ.9) RETURN

```
40  CONTINUE
    RETURN 3
    END
```



## CHAPTER 10

2. For a program which reads in the year (NYEAR), and needs to use the day of the week (NDAY) on which January 1 falls, add:

```
LAST = MOD(NYEAR, 100)
NDAY = LAST + LAST/4
IF (MOD(NYEAR,4).NE.0) NDAY = NDAY + 1
NDAY = NDAY + 1
NDAY = MOD(NDAY,7)
IF (NDAY .EQ. 0) NDAY = 7

5. INTEGER TWO(20,40), ONE(800), SUM
EQUIVALENCE ( TWO(1,1), ONE(1) )
DO 20 I = 1, 20
    DO 20 J = 1, 40
        TWO(I,J) = I**2 + J**2
20 CONTINUE
SUM = 0
DO 30 I = 1, 800
    SUM = SUM + ONE(I)
30 CONTINUE
AVER = SUM/800.0

8. PROGRAM CONVRT
CHARACTER R(10)
INTEGER ZERO
ZERO = ICHAR('0')
READ 4, R
4 FORMAT(10A1)
A = 0.0
DO 10 I = 1, 10
    IF (R(I).EQ.'.') N = I
10 CONTINUE
K = N-1
MULT = 1
20 IF (K.NE.0 .AND. R(K).NE.' ') THEN
    A = A + ( ICHAR(R(K)) - ZERO ) * MULT
    MULT = MULT*10
```

## 634 ANSWERS TO SELECTED EXERCISES

```

        K = K - 1
        GO TO 20
    ENDIF
    K = N + 1
    DIV = 0.1
30 IF (K.LE.10 .AND. R(K).NE.' ') THEN
    A = A + ( ICHAR(R(K)) - ZERO) * DIV
    DIV = DIV/10.0
    K = K + 1
    GO TO 30
ENDIF
PRINT*, R, ' IS', A

```

We have assumed that the value is positive. If it might be negative, we should also search for a ' - ', and if we find one, not examine characters in that position or earlier (that is, say it is in column L, make the test in line 20, K.GT.L), and at the end of the rest of the calculations, set A to -A (that is, use a logical flag):

```

IF (MINUS) A = -A

12. PROGRAM LONGDV
    INTEGER N, M, DIGIT(1000), REM, INTQUO, INTQ, NUM
    PRINT*, 'ENTER TWO INTEGERS TO BE DIVIDED'
    PRINT*, 'AND THE NUMBER OF DECIMAL PLACES YOU WANT'
    READ*, N, M, NUM
***** FIRST TAKE INTEGER QUOTIENT (IF ANY) *****
    INTQUO = N/M
    REM = N - INTQUO*M
    DO 20 I = 1, NUM
        REM = REM*10
        INTQ = REM/M
        DIGIT(I) = INTQ
        REM = REM - INTQ*M
20 CONTINUE
    PRINT 22, N, M, INTQUO, (DIGIT(I), I = 1, NUM)
22 FORMAT('0', I6,'/',I6,' = ', I5,'.',110I1/(1X, 132I1))
    STOP
    END

15. SUBROUTINE OLYMP ( SCORES, N )
    INTEGER SCORES(N), LOW, HIGH
    LOW = COPY(1)
    HIGH = COPY(1)
    DO 50 I = 1, N

```

```

        IF (COPY(I) .LT. LOW) THEN
            LOW = COPY(I)
        ELSEIF (COPY(I) .GT. HIGH) THEN
            HIGH = COPY(I)
        ENDIF
50    CONTINUE
        SUM = 0.0
        DO 60 I = 1, N
            SUM = SUM + COPY(I)
60    CONTINUE
        SUM = SUM - HIGH - LOW
        AVER = SUM/(N-2)
        PRINT 66, AVER
66    FORMAT('0 THE AVERAGE SCORE IS ', F7.2)
        RETURN
        END

```

**20.** PROGRAM SAMEYR

```

INTEGER MONTH, M1, D1, Y1, M2, D2, DAYS
DATA MONTH/31,28,31,30,31,30,31,31,30,31,30,31/
PRINT*, 'ENTER TWO DATES, THE EARLIER ONE FIRST'
PRINT*, 'IN THE FORM MM/DD/YY, ONE SPACE BETWEEN THEM'
READ 5, M1, D1, Y1, M2, D2, Y1
5 FORMAT (6 (I2,1X) )
IF (MOD(Y1,4).EQ.0) MONTH(2) = 29
IF (M1 .EQ. M2) THEN
    DAYS = D2 - D1
ELSE
    DAYS = MONTH(M1) - D1
    DO 20 I = M1 + 1, M2 - 1
        DAYS = DAYS + MONTH(I)
20    CONTINUE
    DAYS = DAYS + D2
ENDIF
PRINT 22, DAYS, M1, D1, M2, D2, Y1
22    FORMAT('0 THERE ARE',I3,' DAYS BETWEEN ',I2,'/',I2,
& ' AND ',I2,'/',I2, ' IN THE YEAR 19',I2)
STOP
END

```

**24.** PROGRAM SPARSE

```

INTEGER LOC(100,2), VAL(100), NUM
PRINT*, 'HOW MANY NON-ZERO VALUES ARE THERE?'
READ*, NUM

```

```

DO 20 I = 1, NUM
    PRINT*, 'ENTER THE ROW AND COLUMN OF EACH NON-ZERO'
    PRINT*, 'VALUE, FOLLOWED BY THE VALUE'
    READ*, LOC(I,1), LOC(I,2), VAL(I)
20 CONTINUE

```

This code would fill a reference array with the locations and nonzero values for such a large array. Now suppose that there are two such arrays that have been filled (LOC1, VAL1, and LOC2, VAL2, and the number of entries in each have been specified as NUM1 and NUM2), as well as the size of the whole array ( $M \times N$ ).

```

SUBROUTINE SPARE (LOC1,VAL1,NUM1,LOC2,VAL2,NUM2,ARR,M,N)
REAL VAL1(NUM1), VAL2(NUM2), ARR(M,N)
INTEGER LOC1(NUM1,2), LOC2(NUM2,2)
DO 10 I = 1, M
    DO 10 J = 1, N
        ARR(I,J) = 0.0
10 CONTINUE
DO 20 L = 1, NUM1
    I = LOC1(L, 1)
    J = LOC1(L, 2)
    IF (I.LT.1.OR.I.GT.M.OR.J.LT.1.OR.J.GT.N) THEN
        PRINT*, 'ERROR IN SUBSCRIPT - ABORT PROGRAM'
        RETURN
    ENDIF
    ARR(I,J) = VAL1(L)
20 CONTINUE
DO 30 L = 1, NUM2
    I = LOC2(L, 1)
    J = LOC2(L, 2)
    IF (I.LT.1.OR.I.GT.M.OR.J.LT.1.OR.J.GT.N) THEN
        PRINT*, 'ERROR IN SUBSCRIPT - ABORT PROGRAM'
        RETURN
    ENDIF
    ARR(I,J) = ARR(I,J) + VAL2(L)
30 CONTINUE
OPEN (3, FILE = 'BIG', STATUS = 'NEW')
DO 40 I = 1, M
    WRITE(3,*) (ARR(I,J), J = 1, N)
40 CONTINUE
CLOSE 3
END

```

```

28. SUBROUTINE CHECK (MEAS, N)
    INTEGER MEAS(N), ME, REM, SUM
    DO 20 I = 1, N
        ME = MEAS(I)
        DO 10 J = 1, 9
            REM = MOD(ME,10)
            SUM = SUM + REM
            ME = ME/10
10      CONTINUE
        IF (MOD(SUM,2).EQ.0) MEAS(I) = MEAS(I) + 1000000000
20      CONTINUE
    RETURN
    END

```

## CHAPTER II

```

3. PROGRAM FIB
***** FIBONACCI NUMBERS *****
    INTEGER FIB(500,20), SUM, CARRY
    DATA FIB/10000*0/
    FIB(1,20) = 1
    FIB(2,20) = 1
    N = 1
    PRINT 11, 1, 1
    PRINT 11, 2, 1
    DO 20 I = 3, 100
        CARRY = 0
        DO 10 J = 1, N
            K = 21 - J
            SUM = FIB(I-1,K) + FIB(I-2,K) + CARRY
            CARRY = 0
        ***** SAVE 9-DIGIT "CHUNKS" *****
        IF (SUM .GT. 999999999) THEN
            CARRY = SUM/1000000000
            SUM = SUM - CARRY*1000000000
        ENDIF
        FIB(I,K) = SUM
10      CONTINUE
        IF (CARRY.GT.0) THEN
            N = N + 1
            FIB(I, 21-N) = CARRY
        ENDIF

```

## 638 ANSWERS TO SELECTED EXERCISES

```
        PRINT 11, I, (FIB(I,J), J = 21-N, 20)
11      FORMAT('0',I3,3X,I9,13I9.9)
20      CONTINUE
      STOP
      END
```

**12.** PROGRAM CORES  
CHARACTER A\*12  
PRINT\*, INPUT AN INTEGER VALUE  
READ\*, N  
WRITE (A,3) N  
3 FORMAT(I12)  
I = 1  
5 IF (A(I:I).EQ.' ') THEN  
 I = I + 1  
 GO TO 5  
ENDIF  
NDIGS = 13 - I  
PRINT\*, 'THERE ARE ', NDIGS, ' DIGITS IN ', N

You may recall the more difficult method of determining the number of digits in an integer by successively dividing it by 10 until the quotient is zero, and counting the number of divisions.

**14.** PROGRAM CENSOR  
CHARACTER\*4 BAD(40), LINE\*80, SHORT\*4  
LOGICAL FOUND  
DATA BAD/ 'DARN', 'RATS', . . . /  
OPEN (3, FILE = 'WHOOPS', STATUS = 'OLD')  
OPEN (4, FILE = 'CLEAN', STATUS = 'NEW')  
5 READ (3, 11, END = 30) LINE  
11 FORMAT(A80)  
DO 20 I = 1, 80  
 IF (LINE(I:I).EQ.' ' .OR. I.EQ.1) THEN  
 K = I + 4  
 IF (K.GT.80) GO TO 25  
 IF (LINE(K:K).EQ.' ' .OR. LINE(K:K) .EQ. '.' .OR.  
& LINE(K:K) .EQ. ',' .OR. LINE(K:K) .EQ. ';' .OR.  
& LINE(K:K) .EQ. ':' ) THEN  
 SHORT = LINE(I:I+3)  
 J = 1  
 FOUND = .FALSE.  
15 IF (J.LE.40.AND..NOT.FOUND) THEN  
 IF(SHORT.EQ.BAD(J)) FOUND = .TRUE.

```

        J = J + 1
        GO TO 15
    ENDIF
    IF (FOUND) LINE(I:I+3) = '****'
    ENDIF
    ENDIF
20  CONTINUE
    GO TO 5
25  WRITE (4, 11) LINE
30  CLOSE 3
    CLOSE 4
    STOP
END

```

**16. PROGRAM PLOTS**

```

***** SINE AND COSINE CURVES *****
DO 20 A = 0.0, 6.28, .1
    X = SIN(A)
    Y = COS(A)
    N = 52 + 50*X
    M = 52 + 50*Y
    PRINT 6
6     FORMAT('0', T52,'|', T<N>, 'S',T<M>,'C')
20  CONTINUE
    STOP
END

```

**18. CHARACTER NAME\*14, UNDER\*14**

```

DATA NAME, UNDER / 'MICHAEL JORDAN', '_____'
PRINT 4, NAME
4  FORMAT('0', 5X, A14)
PRINT 5, UNDER
5  FORMAT('+', 5X, A14)
END

```

**CHAPTER 12****5. PROGRAM PARLIN**

```

CHARACTER LINE(101), C
FILL = 100./6.
PRINT 5, (X, X = 0, 6.0, .24)
5  FORMAT('1',50X, 'PLOT OF Y**2 = 6X (*) AND 2X = Y+6 (#) /'
& 12X,'X', 26F4.1 // 7X, 'Y')

```

```

Y = 6.0
DO 40 I = 1, 81
    LINE(1) = '|'
    C = ' '
    IF (I.EQ.41) C = '-'
    DO 20 J = 2, 101
        LINE(J) = C
20    CONTINUE
P = Y**2/6.0
X = (Y + 6.0)/2.0
M = P*FILL + 1
LINE(M) = '*'
N = X*FILL + 1
LINE(N) = '#'
PRINT 22, Y, LINE
22    FORMAT(6X, F5.2, 5X, 101A1/16X, '|')
Y = Y - .15
40    CONTINUE
STOP
END

```

**8.** PROGRAM SINES

```

CHARACTER LINE(101)
CONV = 3.14159/360.0
DO 40 I = 1, 3
    DO 30 DEG = 0., 360, 20.
        RAD = DEG*CONV
        X = SIN(RAD)
        DO 10 J = 1, 101
            LINE(J) = ' '
10    CONTINUE
        LINE(51) = '|'
        M = 51 + 50*X
        IF (M.LT.51) THEN
            DO 15 K = M, 50
                LINE(K) = '*'
15    CONTINUE
        ELSE
            DO 20 K = 52, M
                LINE(K) = '*'
20    CONTINUE
        ENDIF
        PRINT 22, LINE
22    FORMAT('0', 5X, 101A1)

```



```

        M = 51 + 35*X
        DO 30 J = 1, 101
            LINE(J) = ' '
30      CONTINUE
            LINE(51) = '|'
            LINE(M) = '*'
            PRINT 33, LINE
33      FORMAT('0', 5X, 101A1)
40      CONTINUE
50      CONTINUE
        TPI = 2.0/PI
***** SAWTOOTH WAVE *****
        DO 150 I = 1, 20
            PRINT 11, I
            DO 140 A = 0.0, 2.0*PI, 0.2
                SUM = 0.
                M = 1
                DO 120 K = 1, I
                    N = 2*K - 1
                    SUM = SUM + M*SIN(K*A)/K
                    M = M*(-1)
120      CONTINUE
                X = TPI * SUM
                M = 51 + 45*X
                DO 130 J = 1, 101
                    LINE(J) = ' '
130      CONTINUE
                    LINE(51) = '|'
                    LINE(M) = '*'
                    PRINT 33, LINE
140      CONTINUE
150      CONTINUE
        STOP
        END
    
```

- 15.** This makes a sort of “device driver,” that must operate under certain restrictions (such as a limited width for the picture, so that it can be sent to a laser printer), and which will take any time and output the “watch” with that time.

```

PROGRAM TIME
INTEGER HOUR, MIN
READ*, HOUR, MIN
CALL WATCH (HOUR, MIN)
STOP
END
    
```

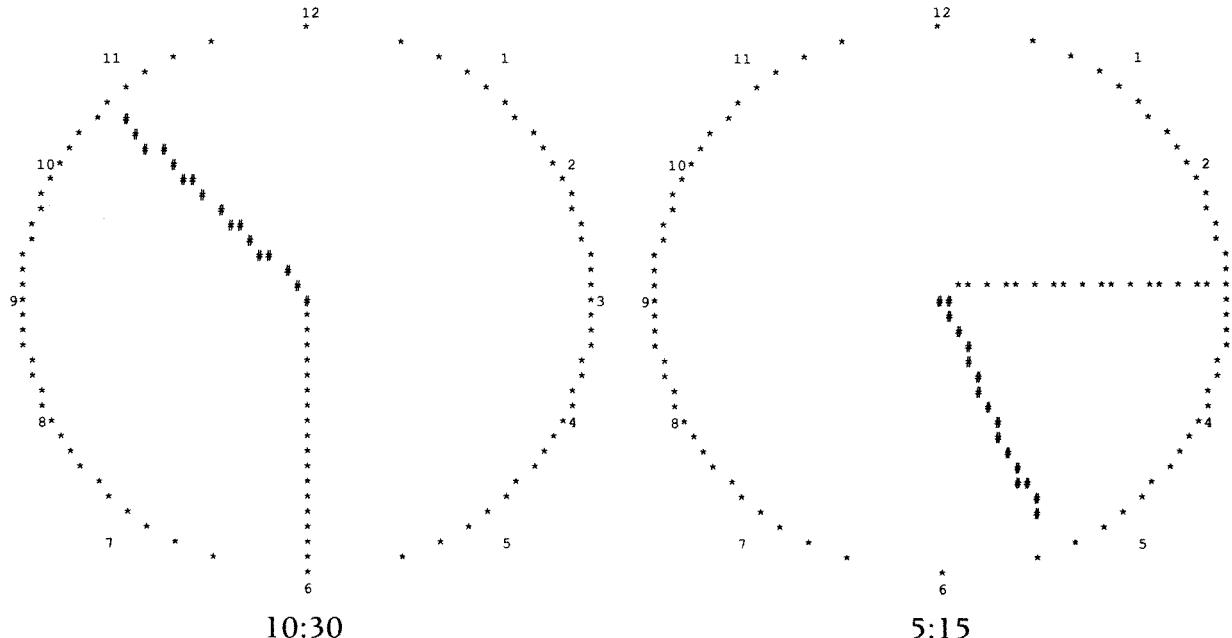
```
SUBROUTINE WATCH (HOUR, MIN)
INTEGER HOUR, MIN, R1, R2
CHARACTER C(37,61)
DATA C /2257*'  /
S = 5.0/3.0
PI2 = 2*3.14159
DO 10 I = 1, 37
    Y = 19. - I
    X = SQRT(324. - Y**2)
    J = 31.49 + X*S
    K = 31.49 - X*S
    C(I,J) = '*'
    C(I,K) = '*'
10 CONTINUE
C(19,31) = '#'
C(3,11) = '1'
C(3,10) = '1' C(3,52) = '1'
C(10,3) = '1'
C(10,4) = '0'
C(10,59) = '2'
C(27,3) = '8'
C(27,59) = '4'
C(35,10) = '7'
C(35,52) = '5'
THETA1 = (HOUR/12. + MIN/720.)*PI2
THETA2 = MIN/60.*PI2
R1 = 16
R2 = 18
CALL POINTS (R1, S, THETA1, C, '#')
CALL POINTS (R2, S, THETA2, C, '*')
PRINT ('''1'', 34X, '''12'')
DO 20 I = 1, 18
    PRINT 25, (C(I,J), J = 1,61)
20 CONTINUE
25 FORMAT(5X, 61A1)
    PRINT 26, (C(19,J), J = 1, 61)
26 FORMAT(4X, '9', 61A1, '3')
    DO 30 I = 20, 37
        PRINT 25, (C(I,J), J = 1, 61)
30 CONTINUE
    PRINT '(35X,''6'')
    RETURN
END
SUBROUTINE POINTS (R, S, THETA, C, Z)
CHARACTER C(37,61), Z
```

```

INTEGER R
DO 50 I = 1, R
    Y = I*COS(THETA)
    X = I*SIN(THETA)
    II = 31.49 + X*S
    J = 19 - Y
    C(J, II) = Z
50 CONTINUE
RETURN
END

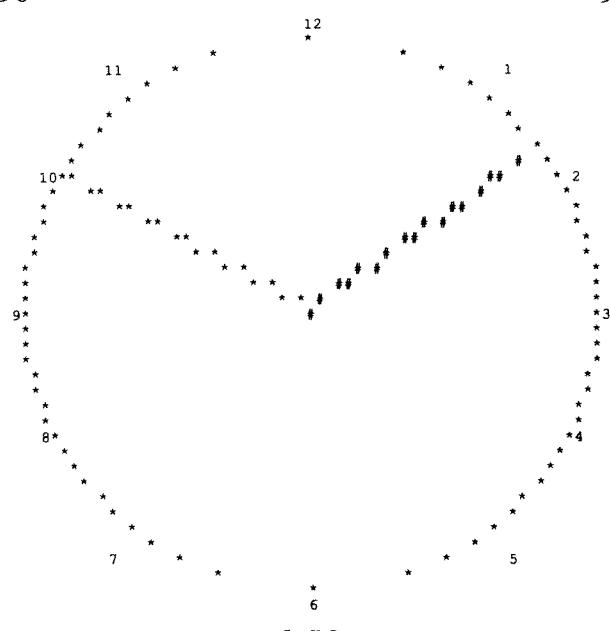
```

Some outputs from this program follow:



10:30

5:15



1:50

```

22. PROGRAM SLOT
CHARACTER*3 ARR(4), A(4)
DATA ARR/'+++', '###', '%%%', '$$$'/
READ*, NSEED
PRINT 5
5 FORMAT('1', ' ONE ', ' TWO ', ' THREE ', ' FOUR'//)
DO 30 I = 1, 20
    DO 20 J = 1, 4
        N = 4*RAN(NSEED) + 1
        A(J) = ARR(N)
20    CONTINUE
***** DELAY LOOP *****
DO 25 K = 1, 100000000
    DO 25 M = 1, 10000
25    MM = 5*2**3**2 + 8**4 - 2.3**5*6
    PRINT 8, (A(J), J=1,4)
8     FORMAT('+', 4(2X, A3, 2X) )
    IF (A(1).EQ.A(2).AND.A(2).EQ.A(3).AND.A(3).EQ.A(4))THEN
        PRINT*, '    WINNER!!!'
        STOP
    ENDIF
30    CONTINUE
END

```

Of course, the program could be modified not to stop after a winner, by moving the STOP to after 30 CONTINUE.

## CHAPTER 13

```

2. PROGRAM JOHNNY
READ*, N
DO 30 I = 1, 100
30    PRINT*, RAND(N), N
FUNCTION RAND(N)
NN = N*N
N = MOD(NN/100, 10000)
RAND = N/10000.
RETURN
END

```

*Note:* This program ended up in a tight loop of values on several different seeds that were tried.

**6. PROGRAM ARAMIS**

```

NP = 0
NA = 0
N = 123451
DO 20 I = 1, 100
    P = RAN(N)
    IF (P .LT. 0.25) THEN
        NA = NA + 1
    ELSE
        A = RAN(N)*2.5
        IF (A .LT. 1.0) THEN
            NP = NP + 1
        ELSE
            GO TO 10
        ENDIF
    ENDIF
20 CONTINUE
PRINT*, 'ARAMIS SHOT', NA, ' TIMES'
PRINT*, 'PORTHOS SHOT', NP, ' TIMES'
END

```

**11. PROGRAM LOADED**

```

COMMON SEED
INTEGER SEED, LOSS, NWIN, ONE, TWO
READ*, SEED
LOSS = 0
NWIN = 0
DO 100 I = 1, 100
    CALL ROLL (ONE)
    IF (ONE.EQ.2.OR.ONE.EQ.3.OR.ONE.EQ.12) THEN
        LOSS = LOSS + 1
    ELSEIF (ONE.EQ.7.OR.ONE.EQ.11) THEN
        NWIN = NWIN + 1
    ELSE
        CALL ROLL (TWO)
        IF (TWO.EQ.7) THEN
            LOSS = LOSS + 1
        ELSEIF (TWO. EQ. ONE) THEN
            NWIN = NWIN + 1
        ELSE
            GO TO 50
        ENDIF
    ENDIF
100 CONTINUE

```

```

PRINT*, NWIN, ' WINS AND', LOSS, ' LOSSES'
STOP
END
SUBROUTINE ROLL (SUM)
INTEGER SEED, SUM
REAL WEIGHT(6)
DATA WEIGHT / 0.2, 0.35, 0.5, 0.65, 0.8, 1.0 /
SUM = 0
DO 5 I = 1, 2
  X = RAN(SEED)
  DO 3 J = 1, 6
    IF (X.LT.WEIGHT(J)) GO TO 4
3   CONTINUE
4   SUM = SUM + J
5   CONTINUE
RETURN
END

```

## 12. PROGRAM INFO

```

CHARACTER C(27), PAGE(2000)
REAL WEIGHT (27), CUM(27)
DATA WEIGHT / 0.1859, 0.0642, 0.0217, . . . /
READ*, NSEED
CUM(1) = WEIGHT(1)
DO 10 I = 2, 27
  CUM(I) = WEIGHT(I) + CUM(I-1)
10 CONTINUE
C(1) = ' '
NREF = ICHAR('A')
DO 20 I = 1, 27
  C(I) = CHAR(NREF)
  NREF = NREF + 1
20 CONTINUE
DO 40 J = 1, 2000
  X = RAN(NSEED)
  DO 30 K = 1, 27
    IF (X .LT. CUM(K)) GO TO 35
30 CONTINUE
35 PAGE(J) = C(K)
40 CONTINUE
PRINT 66, PAGE
66 FORMAT(' ', 50A1)
STOP
END

```

**14.** PROGRAM TDATA  
 INTEGER SEED  
 REAL A, B  
 PRINT\*, 'ENTER A SEED FOR THE RANDOM NUMBER GENERATOR'  
 READ\*, SEED  
 PRINT\*, 'ENTER THE VALUE RANGE YOU WANT TO CREATE'  
 PRINT\*, 'LOWER LIMIT OF THE RANGE FIRST, THEN UPPER LIMIT'  
 READ\*, A, B  
 OPEN (3, FILE='TDATA.DAT', STATUS = 'NEW')  
 DO 20 I = 1, 1000  
 VALUE = (B-A)\*RAN(SEED)  
 WRITE (3, \*) VALUE  
 20 CONTINUE  
 CLOSE (3)  
 STOP  
 END

**26.** PROGRAM BROWN  
 INTEGER SEED, X, Y  
 CHARACTER PLOT (0:99, 0:99)  
 READ\*, SEED  
 X = 100\*RAN(SEED)  
 Y = 100\*RAN(SEED)  
 PLOT(Y,X) = '#'  
 DO 20 I = 1, 50  
 5 XDIFF = 5\*RAN(SEED) + 1  
 XP = X + XDIFF  
 IF (XP.GT.99 .OR. XP.LT.0) GO TO 5  
 6 YDIFF = 5\*RAN(SEED) + 1  
 YP = Y + YDIFF  
 IF (YP.GT.99 .OR. YP.LT.0) GO TO 6  
 CALL LINE (X, Y, XP, YP, PLOT)  
 X = XP  
 Y = YP  
 20 CONTINUE  
 DO 30 J = 0, 99  
 PRINT 33, (PLOT(J,K), K = 0,99)  
 33 FORMAT(5X, 100A1)  
 30 CONTINUE  
 STOP  
 END  
 SUBROUTINE LINE (X, Y, XP, YP, PLOT)  
 CHARACTER PLOT (0:99, 0:99)

{Write a routine here to draw a "straight" line between (X,Y) and (XP, YP), analogous to the POINTS subroutine in 12-15.}

```

28. PROGRAM ART
CHARACTER C(100,100)
DATA C / 10000*' ' /
READ*, NSEED
DO 20 I = 1, 1000
    M = 100*RAN(NSEED) + 1
    N = 100*RAN(NSEED) + 1
    C(M,N) = '*'
20 CONTINUE
DO 30 I = 1, 100
    PRINT 33, (C(I,J), J = 1, 100)
33     FORMAT(4X, 100A1)
30 CONTINUE
END

33. SUBROUTINE INSERT (X, N)
***** BRIDGE-PLAYER INSERTION SORT *****
INTEGER X(N) , XERT, TEMP
DO 20 I = 2, N
    XERT = X(I)
    DO 10 J = I-1, 1, -1
        IF (XERT .GE. X(J)) GO TO 20
        TEMP = X(J)
        X(J) = XERT
        X(J+1) = TEMP
10     CONTINUE
20 CONTINUE
RETURN
END

34. PROGRAM FRACTL
CHARACTER C(0:100, 0:100)
READ*, M, N
NS = 12345657
DO 5 I = 0, 100
    DO 5 J = 0, 100
        C(I,J) = ' '
5     DO 400 I = 1, 100000
        Q = RAN(NS)*3
        IF (Q.LT.1) GO TO 10
        IF (Q.LT.2) GO TO 20

```

```

        M = (M + 100)/2
        N = (N + 100)/2
        GO TO 100
10      M = M/2
        N = N/2
        GO TO 100
20      M = M/2
        N = (N+100)/2
100     IF (I.LT.1000) GO TO 400
        C(M,N) = '*'
400     CONTINUE
        DO 500 I = 100, 0, -1
            PRINT 66, (C(I,J), J = 0, 100)
66      FORMAT(' ', 101A1)
500     CONTINUE
        END
    
```

## ◆ CHAPTER 14

3. SUBROUTINE SMOOTH (A)  
 REAL A(100)  
 DO 20 I = 2, 99  
 A(I) = (A(I-1) + A(I) + A(I+1))/3.0  
20 CONTINUE  
 RETURN  
 END
- \*\*\*\*\* SUBROUTINE TO SMOOTH ONLY ORIGINAL VALUES \*\*\*\*\*  
 SUBROUTINE SMOOCH (A)  
 REAL A(100), B(100)  
 DO 10 I = 1, 100  
10 B(I) = A(I)  
 DO 20 I = 2, 99  
 A(I) = (B(I-1) + B(I) + B(I+1))/3.0  
20 CONTINUE  
 RETURN  
 END
6. PROGRAM NOISE  
 REAL B(50,50), T  
 READ\*, ((B(I,J), J = 1,50), I = 1, 50)  
 READ\*, T  
 DO 20 I = 3, 47

```
DO 20 J = 3, 47
    IF (B(I,J).GT. T) THEN
        SUM = - B(I,J)
        DO 15 II = I-2, I+2
            DO 15 JJ = J-2, J + 2
                SUM = SUM + B(I,J)
            B(I,J) = SUM/24.0
15      CONTINUE
        STOP
    END

15.   FUNCTION ROOT (A, N)
    DATA EPSI /0.0001/
    K = N - 1
    Y = X/N
5     CONTINUE
    XP = (K*Y + X/Y**K)/N
    DIFF = ABS(Y/XP - 1.0)
    Y = XP
    IF (DIFF .GE. EPSI) GO TO 5
    ROOT = XP
    RETURN
END

17.   FUNCTION POWER (X, N)
    PROD = 1.0
    XX = X
    K = N
5     IF (K.GT.0) THEN
        IF(MOD(K,2).EQ.1) PROD = PROD*XX
        K = K/2
        XX = XX*XX
        GO TO 5
    ENDIF
    POWER = PROD
    RETURN
END
```

## CREDITS FOR CHAPTER OPENING PHOTOGRAPHS

|              |                                                                                                                                                                                                                                                           |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Introduction | NASA                                                                                                                                                                                                                                                      |
| Chapter 1    | TM & C COURTESY OF LUCASFILM LTD. (LFL) 1980.<br>All Rights Reserved.                                                                                                                                                                                     |
| Chapter 2    | Gary Fong/San Francisco Chronicle                                                                                                                                                                                                                         |
| Chapter 3    | NASA                                                                                                                                                                                                                                                      |
| Chapter 5    | IBM, Burlington, Vermont. Computer generated plot<br>on paper of Dynamic Random-Access Memory Chip<br>(DRAM). 1982. 288,000 transistors. Collection, The<br>Museum of Modern Art, New York. Gift of the<br>manufacturer, courtesy of the IBM Corporation. |
| Chapter 6    | IBM                                                                                                                                                                                                                                                       |
| Chapter 7    | The National Radio Astronomy Observatory, operated<br>by Associated Universities, Inc., under contract with<br>the National Science Foundation.                                                                                                           |
| Chapter 8    | NASA                                                                                                                                                                                                                                                      |
| Chapter 9    | Travis Amos                                                                                                                                                                                                                                               |
| Chapter 10   | Bridgeman/Art Resource, NY                                                                                                                                                                                                                                |
| Chapter 11   | Seagate Technology                                                                                                                                                                                                                                        |
| Chapter 12   | Dennis di Cicco                                                                                                                                                                                                                                           |
| Chapter 13   | Michael McKenna. Copyright 1990 MIT Media Lab.                                                                                                                                                                                                            |
| Chapter 14   | NASA                                                                                                                                                                                                                                                      |
| Chapter 15   | Cray Research, Inc.                                                                                                                                                                                                                                       |

# INDEX

- Abacus, 2  
Absolute value, 87  
    of a complex number, 208  
Abstract data type (ADT), 483–488  
ACCESS=, 406, 409  
Accumulation, 119–120  
Ada, Lady Lovelace, 2  
Air quality standards, 136–137, 295–296  
Airplane:  
    in crosswind, 89  
    two planes on collision (?) course, 456  
Algebraic expressions (translated into  
    FORTRAN), 70–71  
Algorithm, 25–28, 37–38  
Allometric relationship (ecology), 534  
ALU (Arithmetic and Logic Unit), 5–6  
American National Standards Institute  
    (ANSI), 12–13  
American Standards Association (ASA), 15  
“Amicable” numbers, 358  
Analogy, 35  
Analytical Engine, 2–3  
AND operator, 107–108, 111  
Area between two curves, 521  
Areas enclosed by rope, 36–40  
Arguments (dummy and actual), 325,  
    330–333, 344–349  
Arithmetic IF, 105–107, 588–589  
Arithmetic operators, 55–61, 64–66  
Array operations (Fortran 90), 602–603  
Array segments (Fortran 90), 602  
Arrays, 241–297  
    one-dimensional, 242–252  
    two-dimensional, 262–268  
    higher-dimensional arrays, 268  
ASCII, 7, 239, 593–595  
Assembly language, 9–11, 19  
Assigned GO TO, 372, 584, 588  
Assignment statement, 54–61, 71, 583–584  
Atanasoff, John V, 3  
Babbage, Charles, 2  
BACKSPACE, 407–408, 588  
Backus, John, 14  
Bank interest, 173–174  
Barbarian and Roman soldier, 129–130  
Baseball pitcher’s throw, 89  
Basketball tryout roster, 122–123  
Batch mode, 13  
Billings, John, 2  
Binary arithmetic, 575–576  
Binary representation, 6–9, 18, 569–575  
Biorhythms, 439–442, 453  
Birthday probability, 504  
Bit, 6–8  
Blanks handling:  
    BN and BZ, 228, 402  
    BLANK= clause, 406, 410  
BLOCK DATA 339–340, 584  
Block If (IF/THEN), 100–101, 589  
Block letters, 84  
Block structure, 542–543  
Blood donors, 239  
Blue Moon, 506  
Bode’s Law, 176–177  
Bouncing (tennis) ball, 161–162, 173  
Brownian motion, 503  
Bubble sort, 273–276, 290  
Byte, 8  
Calendar program, 292, 391–392  
CALL, 168, 330, 584  
Calorie, 85  
Cancellation errors, 312  
Cards, punched, 7  
Carriage control, 212–215  
Cartesian graphs, 442–448  
Case. *See* Select Case  
Cell. *See* Location  
Center of gravity, 207, 394, 500–501  
CHAR function, 191–192

Character constants, data type and variables, 50, 52–54, 186–188  
 “Check” digit, 397  
 Chess programs, 358–359, 363  
 Circuit board, 208–209  
 Clock time (analog display), 456  
 CLOSE, 408–409, 588  
 Clown shot out of cannon, 70–71  
 Cocktail shaker sort, 276  
 Collating sequence, 191  
 Combinations, 254–256, 294  
 Comments, 30, 74, 82, 585  
 COMMON, 337–343, 585  
     with EQUIVALENCE, 377–379  
     blank COMMON, 337–338  
     labelled COMMON, 338–339  
 Compatibility, 15–16  
 Compiler, 12  
 Complex constants, data type and variables, 50, 52–54, 192–196  
 Complex roots, 206  
 Computed GO TO, 368–372, 588  
 Computers, brief history of, 2–3  
 Concatenation, 189  
 Conditional branch. *See* Logical IF, Block IF, IF/THEN/ELSE, Arithmetic IF  
 Conservation of momentum, 136, 137  
 Constants, 46–50  
 CONTINUE, 146, 546, 585  
 Control statements, 91–137  
 Convolution, 516  
 Conway, John, 501–502  
 Core-to-core data transfer. *See* Internal files  
 Correlation, 425  
 Counter, 117–118  
     arrays for counters, 271–272  
 CPU (Central Processing Unit), 5–6  
 Craps (simulating the game of), 392, 498, 499  
 Cryptography, 395–396  
 Curve fitting. *See* Least squares  
 CYCLE (in Fortran 90), 153  
 Cycloid, 455–456  
 Data abstraction, 483–488  
 Data improvement, 515–516, 530  
     enhancement, 515–516  
     filtering, 516  
     smoothing, 515  
 DATA statement, 199–201, 585  
     initialization of arrays, 280–281  
 Data types and typing, 181–209  
 Day of week determination, 390–391, 395  
 Decision statements. *See* Block IF, IF/THEN/ELSE, Logical IF, Arithmetic IF  
 Default typing, 52–54  
 Defensive programming, 309–310  
 DeMorgan’s Laws, 110, 203, 422  
 Density Plots, 453–454  
 “Deprecated” features (of F77), 601–602  
 Derived data types (Fortran 90), 487–488  
 Determinants, 527–528  
 Differentiation, 522–523  
 Digital logic circuit design, 427–428  
 Dimension, 242, 586  
 DIRECT =, 409  
 Direct-access files, 404–411  
 Directed graphs, 506  
 Documentation, 545, 549–551  
 DOTPRODUCT function (Fortran 90), 603  
 Double-precision constants, data types and variables, 50, 52–54, 196–197  
 Dulong and Petit, Law, 85  
 EBCDIC, 7, 239, 596–597  
 “Echo” input, 548  
 Eclipse, 457  
 EDVAC, 3  
 ELSE. *See* IF/THEN/ELSE  
 ELSEIF, 102–105, 586, 589  
 END= clause, 226–227, 590  
 ENDFILE, 407, 588  
 END statement, 75, 586  
 ENIAC, 3, 6  
 ENTRY, 361–362, 586  
 EQUIVALENCE, 372–379, 586–587  
     with COMMON, 377–379  
 EQV operator, 109, 111  
 ERR= clause, 226–227, 406–407, 409, 590  
 Error handling, 299–321  
 Exchanging values, 120–121, 387–389  
 Execution-time errors, 303–306  
 EXIST =, 409  
 EXIT (in Fortran 90), 153  
 Exponential distribution, 477, 500  
 Exponential growth, 174  
 Exponentiations (order of), 64–66

- EXTERNAL, 347–349, 587. *See also*  
INTRINSIC
- Factorials, 254  
Fermat number, 66  
Fibonacci numbers, 162–163, 288, 422  
FILE =, 405  
File handling, 404–417, 588  
“Flag” value, 116–117  
Flowcharts, 28–32  
FORM =, 406, 409  
Form (physical layout) of FORTRAN statement, 72–74  
FORMATs, 211–239, 587  
blank editing, 228, 402, 406, 410  
character, 220  
complex, 220–221  
double-precision, 221  
Integer (I), 215–217, 400  
Logical (L), 220  
Real (F,G,E), 217–220  
repeated format patterns, 223–225, 231  
scaling, 401–402  
signs, 400  
slash, 222, 230  
tab, 403–404  
FORMATTED =, 409–410  
FORTRAN (history), 14–17  
FORTRAN II, 15  
FORTRAN III, 15  
FORTRAN IV, 15  
FORTRAN V, 15  
FORTRAN ’66, 15–16  
FORTRAN 77, 15–16, 19  
FORTRAN statement (layout), 72–74  
Fortran 8x, 16  
Fortran 90, 16–17, 19, 81–82, 93, 130,  
147, 152–154, 170, 200–202, 259–260,  
272–273, 286, 343–344, 354, 366–367,  
389, 487–488, 490–491, 528, 598–610  
Four-color theorem, 4  
Fractals, 449–451, 504–505. *See also*  
Julia sets  
Free-field format, 82, 606  
FUNCTIONs, 344–351, 587  
iterative functions, 349–351  
“Game of Life”, 501–502  
Gasoline pump simulation, 222–223, 238  
Gaussian distribution, 476–477, 500  
Gaussian elimination, 524–527  
Generic functions, 577–582  
Global variables. *See* COMMON  
Goldbach’s conjecture, 291  
Golden mean, 178  
GO TO (unconditional branch), 95–100,  
588. *See also* Computed GO TO,  
Assigned GO TO  
Graduated income tax, 42, 134  
Graduated pay scale, 42  
Grand Challenges in Science and  
Engineering, 565–566  
Graphs. *See* Plotting  
Gravitational force problem, 77–79, 86  
Greatest common divisor, 123–126, 361  
“Guess the Number”, 472  
“Hangman” program, 479, 500  
Hardware, 4  
Hash coding, 423–424  
Hierarchy charts. *See* Structure charts  
High-level languages, 11–14  
HIPO diagrams. *See* Structure charts  
Histograms, 207, 248, 353, 434–435  
Hollerith, Herman, 2  
Hypotenuse (statement function), 325–326  
Hypothesis testing, 320  
ICHAR function, 191–192  
IF/THEN/ELSE construct, 101–102  
Impedance, 194–196, 206  
IMPLICIT type statement, 198–199  
IMPLICIT NONE, 82, 606  
Implicit typing. *See* Default typing  
Implied lists (I/O), 250–252, 265–266  
Impossible loop, 144  
IMSL, 567  
INCLUDE (Fortran 90), 608  
INDEX function, 190–191  
Inelastic collision, 137  
Information hiding, 544  
Information theory simulation, 499–500  
INQUIRE (file handling), 409–410, 588  
Installment loan payments program, 79–81  
Integer constants, data type and variables,  
47–48, 52–54, 182–183  
Integer truncation (division), 60–62  
Integration. *See* Numeric integration  
INTENT (IN, OUT, INOUT), in Fortran 90,  
354, 609–610

- Interactive mode, 13  
 Interactive programs, 478–479  
 Interchanging values. *See* Exchanging values  
 Internal files (core-to-core data transfer), 414–416, 424  
**INTRINSIC**, 347–349, 589. *See also EXTERNAL*  
**IOSTAT =**, 407, 409  
 Iteration count (for DO), 145–146  
 Iterative functions, 349–351, 360, 531  
 Julia sets, 457–460  
 Karnaugh maps, 428–431  
 Laplace, 295  
 Large numbers, 383–387, 393, 395  
 Least common multiple, 361  
 Least squares, 516–518  
 Leibniz, G W, 2, 12  
 LEN function, 190  
*Limits to Growth*, 532–533  
 Line spectra (quantum mechanics), 177  
 Linear programming, 534–536  
 Linked lists, 493–495  
 List. *See* One-dimensional array, Linked lists  
 List-directed I/O, 66–69, 189–190  
 Location (memory), 46–47  
 Logical constants, data type and variables, 50, 52–54, 184–186  
 Logical IF, 92–95  
 Logical operators. *See* AND, OR, NOT, EQV, NEQV  
 Logical relations, 93  
 Logic errors, 306–309  
 Loops: DO, 140–146, 586  
     DO/END DO, 147, 152–153, 607  
     DO/WHILE, 149–152  
     exits, 148–149  
     nested loops, 154–159  
     prechecked loops, 144–146  
     posttest loops, 114–115  
     Pretest loops, 115–117  
     repeat N times, 112–113  
     repeat/Until loop, 114–115, 149  
         with arrays, 245–248  
 Lower triangular matrix, 506  
 “Machine epsilon”, 177–178  
 Machine language, 9  
 Mandelbrot set, 450–452, 457  
 Mathematical subroutine packages, 567  
**MATMUL** (Fortran 90), 528, 602  
 Matrix multiplication, 267–268  
 Maxima and minima (finding), 268–269  
 MAXLOC and MINLOC procedures (Fortran 90), 602  
 MAXVAL and MINVAL procedures (Fortran 90), 528, 602  
 Measuring inaccessible distances (civil engineering), 208  
 Memory, computer, 5–6  
 Military time, 86  
 Mixed-mode expressions, 62–64  
 Models. *See* Simulation  
 Modularization, 323–363  
 Modules (Fortran 90), 343–344, 604  
 Monetary exchange rates, 394  
 Monte Carlo methods, 473–476  
 Motion effect, 425  
 Motorcycle policeman chasing speeder, 89  
 NAG (Numerical Algorithms Group), 567  
**NAME =**, 409  
**NAMED =**, 409  
**NAMELIST**, 317–318, 608  
 Nassi–Schneiderman diagrams, 33–34  
**NEQV** operator, 109, 111  
 Neural Nets, 508  
 Newton, Sir Isaac, 77  
**NEXTREC =**, 410  
 Normal (Gaussian) distribution, 476–477, 500  
 NOT operator, 109, 111  
**NUMBER =**, 409  
 Number Systems, 569–576  
 Numeric integration, 518–521, 531  
 Numerical centers, 135  
 Object code, 12  
 Object Orientation, 483–484, 509  
 Obsolescent features (of FORTRAN), 599–600  
 Octal and hexadecimal, 571–573  
 Olympics scoring, 394  
 One’s complement, 18  
**OPEN** (in file handling), 405–407, 588  
**OPENED =**, 409  
 Optimization, 561–565  
 OR operator, 107–109, 111

- Overflow, 8, 318
- Packing, 381–383
- Palindrome, 358
- PARAMETER statement, 75–76, 546, 589–590
- Particle accelerator, 87
- Pascal, Blaise, 2
- Pascal’s triangle, 460
- Pattern recognition, 479–482
- Pauling, Linus (on smoking), 87
- PAUSE statement, 590
- Paychecks, writing, 507
- Perfect numbers, 173
- Piano keyboard, 178–179
- Pipelining, 557–559
- Planets (mass, distance from the sun), 77–78
- Plotting, 424, 435–448
- Pointers (Fortran 90), 606
- Pointillist painting, 503
- “Pop”, 490
- Polar coordinates, 205
- Population problems, 175–176
- Portability, 13
- Precision, 49
- Prime factors, 291
- Prime numbers, 160–161, 254–259
- PRINT, 67–69, 212, 590
- Problem clarification, 23–24, 36–37
- Problem solving, 21–43
- Problem specification, 22–23, 36
- PRODUCT procedure (Fortran 90), 603
- PROGRAM statement, 74, 546, 590
- Program verification, 554–555
- Programming team, 555
- Projectiles, 87–88, 136, 175, 357, 455
- Pseudocode, 34–35
- “Push”, 503
- Quadratic equation, roots of, 193–194, 205
- Quadrature, 519–520
- Queues, 491–492, 501
- Radioactive decay, 456, 533
- Random-access files. *See* Direct-access files
- Random data (to test programs), 478
- Random numbers, 465–478
- generators, 466–472
  - in Fortran 90 {RANDOM\_NUMBER(X)}, 469, 495, 609
- linear congruential generator, 467–468, 498
- mid-square method, 466–467
- scaling, 468–472, 504
- Range (of a DO loop), 141–143
- READ, 67–69, 226–230, 590
- Readability Index, 507
- Real constants, data type and variables, 48–50, 52–54, 183–184
- Rebounding (from user errors), 553–554
- RECL =, 406, 410
- Record, 488–489
- Recursion, 359–360, 606
- Reference array, 253–259, 288, 292, 395
- Regression line, 516–518
- Relaxation, 295
- Reliability, 554–555
- Remainder, 61–62
- Remote sensing, 515–516, 536
- Repeat/Until, 114–115, 149
- Repetition, 31–32, 139–179
- Resistors (in parallel), 84, 356–357
- RETURN, 166, 168, 331, 590
- alternate RETURN, 362–363, 590–591
- Revolving charge account, 126–128
- REWIND (in file handling), 407, 588
- “Robust” (program), 309
- Roman numeral to Arabic conversion 396–397
- Roots of equations:
- bisection method, 512–514
  - Newton–Raphson method, 514
- Rotation of graphic objects, 454–455
- “Round” function, 357
- Roundoff errors, 310–311
- SAVE, 354
- Saw-tooth wave, 456
- Scattergram, 426–427
- Scientific notation, 48–49
- Scientific visualization, 433, 448–449
- Screen graphics, 451
- Searching:
- binary, 278–280
  - linear, 269–271
- SELECT CASE (Fortran 90), 366–368, 608–609
- Selection, 29–30. *See also* Arithmetic IF, Block IF, IF/THEN/ELSE, Logical IF
- Selection sort, 276–278

- Sentinel value. *See "Flag" value*  
**SEQUENTIAL** =, 409  
 Sequential files, 404–411  
 Sets, 203–204, 489–490  
 Shell sort, 294  
 Shuffling and dealing cards, 498–499  
 Shuttle/interchange sort, 276–278  
 Sieve of Eratosthenes, 290–291  
 Simulations, 464–478  
 Simultaneous linear equations (solving), 523–527  
 Slot machine simulation, 460  
 Snell's Law, 363  
 Software, 4  
 Software engineering, 551–553  
 Software life cycle, 556  
 Sorting, 273–278. *See also* Shell sort,  
     Bubble sort, Shuttle/interchange sort  
 Sound level classification, 102–105  
 Source code, 12  
 "Spaghetti code", 96  
 "Sparse" matrix, 396  
 Speeding penalties (in Pa ), 135–136  
 Spell-checker, 507  
 Square wave, 456  
 Stacks, 490–491  
 Standard deviation, 236, 246–247  
 Standards for FORTRAN programs, 540–541  
 Statement functions, 324–328, 587  
     restrictions on, 326  
 Static friction, 88–89  
 STATUS=, 405–406  
 Stepwise refinement, 24–35  
 STOP statement, 75, 591  
 Stride (array, in Fortran 90), 602  
 Structure charts, 32–33  
 Structured program design, 541–544  
 Structured walkthroughs, 555  
 Style (programming), 544–548  
 Subroutines, 329–336, 591  
     preview to, 165–170  
 Subscripts, 244–245  
 Substrings, 188–189, 204–205  
 SUM array procedure (Fortran 90), 603  
 Supercomputers, 556–561  
 Symbolic constants. *See* PARAMETER  
 Symbolic debugger, 312  
 Syntax errors, 300–303  
 System functions, 329, 577–582  
 Tab, 403–404  
 Team projects, 555  
 Temperature conversion, 86  
     statement function for, 326–327  
 Testing (program), 35–36, 38  
 Text handling, 379–381, 396  
 Top-down design, 24–25  
 TRANSPOSE procedure (Fortran 90), 528, 603  
 Trapezoidal rule, 520–521  
 Two-body problem, 77–79  
 Two's complement, 18, 396  
 Type statements, 52–53, 591  
 Ulam, Stanislaw, 475, 508  
 Unconditional branch. *See* GO TO  
 Underflow, 318  
 UNFORMATTED =, 410  
 Unformatted I/O, 411–412  
 UNIT =, 405  
 Universe, age of (compared to your age), 84  
 "Unrolling" a loop, 564  
 Updating (a file), 412–414  
 User-defined types, 201, 260–261, 484–488, 604–605  
 Utilitarian decision-making, 294–295  
 Variable dimensioning, 335–336  
 Variable formatting, 417–420  
 Variables, 46–47, 51–54  
 Vector processing, 281–285  
 Vectors. *See* One-dimensional arrays  
 von Neumann, John, 3, 87, 475, 508  
 Water quality, 136  
 Watson, Thomas J. , 3  
 WHERE statement (Fortran 90), 259, 272–273, 602  
 WHILE/DO (or DO/WHILE), 149–152, 608  
 "Window" in graphics, 456–457  
 Wine cask dimensions (optimum), 163–165  
 Wirth, Nicklaus, 14  
 Word. *See* Location  
 Word size (computer), 5–6  
 WRITE, 212, 590  
 Writing to printer and screen, 416–417  
 Zodiac, 392