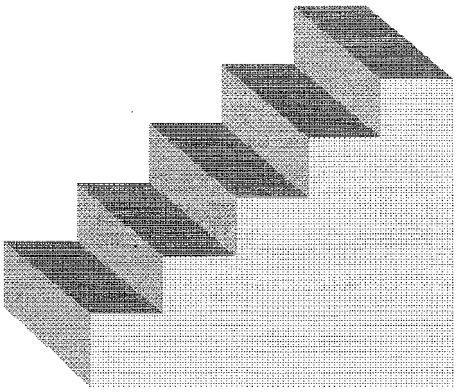


PROGRAMMING IN FORTRAN

Fourth Edition

Ömer Akgiray

PERMISSION TO COPY AND DISTRIBUTE:

e-mail:

omer.akgiray@marmara.edu.tr

CHAPTER 3: *LOOPS AND ARRAYS*

3.1 The GO TO Statement

The statements of a FORTRAN program are normally executed sequentially from top to bottom, as they appear in the program listing. In many applications, however, the sequence of steps to be performed depends on the input data and certain intermediate quantities that are calculated during program execution. Program structures which affect the order in which statements are executed, or which affect whether certain statements are executed or not, are called **control structures**. (When the structure in question consists of a single statement, such as the logical IF statement, we usually speak of a **control statement**.)

We have already seen how the block-IF control structure can be used to make decisions, based upon the values of input and/or calculated data, as to which sequence of program steps is to be performed. Another language element that is often used to control the execution order of the instructions in a FORTRAN program is the GO TO **statement**. The general format of the unconditional GO TO statement is as follows:

GO TO *m*

where *m* is the statement number (label) of the executable statement to which control is transferred. This statement instructs the computer to go, unconditionally, to that part of the program beginning with the statement labeled *m*. A more frequently used construct is a combination of the logical IF statement with the GO TO statement:

IF(*logical expression*) GO TO *m*

This is a conditional transfer statement in that the transfer of control to the statement labeled *m* is contingent on the truth of the *logical expression*. If the expression is false, then the GO TO statement is not executed and execution continues with the first program statement after the conditional GO TO statement.

Example 3.1:

The following program illustrates the use of the GO TO and the logical IF statements.

```
      PROGRAM PARGRM
C*************************************************************************
C   Program calculates the area and the perimeter of a parallelogram
C   with side lengths A and B, and angle THETA between these sides.
C*************************************************************************
```

```
      IMPLICIT NONE
      REAL A, B, THETA, AREA, PER, PI
      PI = ACOS(-1.)
C   Read input data. Check for possible user errors.
      PRINT*, 'Enter length of side A: '
      READ*, A
      IF(A.LE.0.0) GO TO 1
      PRINT*, 'Enter length of side B: '
      READ*, B
      IF(B.LE.0.0) GO TO 2
      PRINT*, 'Enter angle (in degrees) between A and B: '
      READ*, THETA
      IF(THETA.LE.0.0 .OR. THETA.GE.180.0) GO TO 3
C
C   Calculate and display results.
      AREA = A*B*SIN(PI*THETA/180.)
      PER = 2*(A+B)
      PRINT*, 'Area      =', AREA
      PRINT*, 'Perimeter =', PER
      STOP 'Execution successfully completed. '
C
C   Error messages
    1 PRINT*, 'Value entered for side A is ',A
      PRINT*, 'You must enter a positive value.'
      GO TO 4
    2 PRINT*, 'Value entered for side B is ',B
      PRINT*, 'You must enter a positive value.'
      GO TO 4
    3 PRINT*, 'Value entered for angle is ',THETA
      PRINT*, 'You must enter a value between 0 and 180.'
    4 STOP 'Execution terminated.'
      END
```

Here is a sample run:

```
Enter length of side A: 1.0
Enter length of side B: 2.0
Enter angle (in degrees) between A and B: 90.
Area      =    2.00000
Perimeter =    6.00000
Execution successfully completed.
```

The following illustrates what happens in the case of illegal input:

```
Enter length of side A: -1.
Value entered for side A is    -1.00000
You must enter a positive value.
Execution terminated.
```

3.2 Introduction to Loops

Sometimes we want the computer to repeat a portion of a program many times. One obvious--and certainly awkward--way to accomplish this would be to retype the relevant program statements as many times as they need to be executed. In many

applications, however, this approach would necessitate the use of thousands or even millions of program statements. Fortunately, such cumbersome programs never have to be developed, as one of the fundamental capabilities of a computer is **looping**, i.e. its ability to repetitively execute a set of statements.

In this and the following two sections, we shall learn how to implement loops in FORTRAN. The discussion in this section is somewhat informal, our purpose being the introduction, by example, of some of the basic concepts and methods related to the implementation of loops in FORTRAN. A more careful study of loops is presented in the next two sections.

One method of constructing loops in FORTRAN is the use of the `GO TO` statement. This is possible because the statement to which control is transferred may precede the `GO TO` statement in the program. The use of the `GO TO` statement to build a loop is demonstrated in the next example.

Example 3.2:

Suppose we want to produce a table showing equivalent pound and kilogram weights. (1 kg is equal to 2.2046226 lbs.) The program will read in the smallest and the largest weights as well as the difference between successive weights that will be displayed in the table. Here is a FORTRAN program that can be used to construct the desired conversion table:

```
      PROGRAM LBTOKG
C*************************************************************************
C   Program to generate a table of lb to kg conversion.
C   Each kg value is rounded up or down to nearest integer.
C*************************************************************************
      IMPLICIT NONE
      INTEGER LB, KG, MINLB, MAXLB,STEPLB
C
C   Read loop parameters
      PRINT*, 'Enter minimum and maximum weights (in pounds): '
      READ*, MINLB, MAXLB
      PRINT*, 'Enter step size (in pounds): '
      READ*, STEPLB
C
C   Print table headline
      PRINT*, '       Pounds        KGs'
      PRINT*, '       ------        ---'
C
C   Calculate and display conversion table
      LB = MINLB
    1 KG = NINT(LB/2.2046226)
      PRINT*, LB, KG
      LB = LB + STEPLB
      IF(LB.LE.MAXLB) GO TO 1
      END
```

The statements that read in the loop parameters should be clear. Next, the variable `LB` (weight in pounds) is initialized to the first weight that will be displayed in the table. The next statement is the first statement of the loop and carries out the required conversion:

```
1 KG = NINT(LB/2.2046226)
```

We see that the statement number (label) of this statement is 1. (Any number between 1 and 99999 could be used.) The operation within parentheses is real division and its result is therefore real. The function NINT finds the nearest integer to this value and assigns this integer value to the integer variable KG. Next, the result is printed: this generates a single row of the desired table. The value of LB is then incremented and compared with the maximum weight that we want to have displayed:

```
IF(LB.LE.MAXLB) GO TO 1
```

If LB is strictly greater than MAXLB, then the GO TO statement is not executed and the program execution is ended by the END statement. Otherwise, control is transferred back to the statement labeled 1 and the corresponding value of KG is calculated and printed, and LB is again incremented.

The statements of the loop are executed repeatedly as long as LB.LE.MAXLB is true, and the loop terminates as soon as it is found to be false. (Due to the way we have implemented our loop, the body of the loop will be executed at least once, even if the value of LB.LE.MAXLB is never true. Note that the loop condition LB.LE.MAXLB can be false initially only if there is an input error, i.e. if the initial value of LB--i.e. MINLB--is larger than MAXLB.) Each pass through the loop generates a row of the conversion table. Here is a sample output:

```
Enter minimum and maximum weights (in pounds):  100   300

Enter step size (in pounds): 20

       Pounds         KGs
       ------         ---
         100           45
         120           54
         140           64
         160           73
         180           82
         200           91
         220          100
         240          109
         260          118
         280          127
         300          136
```

Remark: To make the program as simple as possible and to keep the focus on the construction of the loop, no error-checking was included in the program. In a real application, however, it is very important that you write code to handle all conceivable user-errors.

The looping method illustrated in the above program is just one of the ways the GO TO statement can be used to implement a loop:

```
1 KG = NINT(LB/2.2046226)
  PRINT*, LB, KG
  LB = LB + STEPLB
  IF(LB.LE.MAXLB) GO TO 1
```

An entirely equivalent but probably a more readable way of implementing this loop involves the use of the **CONTINUE statement** and using indention to clearly delimit the body of the loop:

```
1 CONTINUE
      KG = NINT(LB/2.2046226)
      PRINT*, LB, KG
      LB = LB + STEPLB
  IF(LB.LE.MAXLB) GO TO 1
```

The CONTINUE statement is used here simply to mark the beginning of the loop, and its function is exactly described by its name: *continue execution*. Actually, the CONTINUE statement is most often used to mark the end of a loop, as opposed to its beginning (cf. the discussion of the DO loop in this chapter). Consider next the following loop structure:

```
1 IF(LB.GT.MAXLB) GO TO 2
      KG = NINT(LB/2.2046226)
      PRINT*, LB, KG
      LB = LB + STEPLB
  GO TO 1
2 CONTINUE
```

If the initial value of LB, i.e. MINLB, is less than or equal to MAXLB (as it should be), this loop will accomplish exactly the same task. Yet another alternative approach is seen in the following structure:

```
1 IF(LB.LE.MAXLB)THEN
      KG = NINT(LB/2.2046226)
      PRINT*, LB, KG
      LB = LB + STEPLB
  GO TO 1
  ENDIF
```

In what follows, we shall learn how to implement the last two loops above using the DO WHILE loop structure which is arguably a more elegant loop control structure. However, as is explained later in this chapter, the implementation of certain types of loops in standard FORTRAN 77 necessitates the use of the GO TO statement. It is therefore important for you to understand the various ways in which the GO TO statement can be utilized to construct loop control structures.

3.3 The DO WHILE Loop

The use of the GO TO statement for the construction of loops was described in the previous section. In this section and the next, we shall study two loop control structures--the DO WHILE loop structure and the DO loop structure--which make

possible the development of more readable and structured FORTRAN programs by obviating the need for the GO TO statement in the implementation of certain types of loops. We shall first discuss the **DO WHILE loop** control structure:

```
DO WHILE(logical expression)
     loop body
END DO
```

The *logical expression* is evaluated first. If it is true, then the *loop body* is executed. This sequence (i.e. evaluation of *logical expression* first, execution of *loop body* next) is repeated as long as *logical expression* evaluates true. If the *logical expression* evaluates false, the *loop body* is skipped and execution continues with the first statement following END DO.

The DO WHILE loop control structure was not included in standard FORTRAN 77[1]. As a result, it was not available on some of the older FORTRAN compilers. On some compilers, it used to be available in the slightly different form shown below:

```
WHILE(logical expression) DO
     loop body
ENDWHILE
```

In case the DO WHILE loop is not incorporated in a FORTRAN compiler, one of the following two forms utilizing the GO TO statement can be used instead of the DO WHILE loop structure. The first alternative structure is one that employs the logical IF statement:

```
m   IF(.NOT.logical expression) GO TO n
        loop body
    GO TO m
n   CONTINUE
```

where *m* and *n* are the statement numbers (labels) of the loop header and the loop terminator, respectively. Here the *loop body* is repeated as long as the loop exit condition

```
.NOT.logical expression
```

is false, i.e. as long as *logical expression* is true. Another structure that is equivalent to the DO WHILE loop is the following:

---

[1] Standard Fortran 90 includes the DO WHILE loop and a more general DO-END DO structure.

```
m   IF(logical expression) THEN
        loop body
    GO TO m
    ENDIF
```

Notice how we always indent the block of code (i.e. *loop body*) that is acted upon by the control structure, leaving unindented the structure itself. For example, the two final statements of the above format are not indented, since they are part of the control structure, not of the inside block. Again, this is not required, but improves the readability of the program.

Example 3.3:

The following program provides an example of the use of the DO WHILE statement. The program determines the *gcd* (greatest common divisor) of two integer values. The *gcd* of two integers is the greatest integer that evenly divides the two integers. For example, the *gcd* of 10 and 20 is 10, because 10 is the largest integer that evenly divides both 10 and 20.

An algorithm that can be used to find the *gcd* is as follows: Let M and N be two nonnegative integers. Step 1: If N is zero, then *gcd* is equal to M. Otherwise, execute Step 2: Calculate the remainder of M/N, assign the previous value of N to M, assign the remainder of M/N to N, and go back to Step 1.

```
      PROGRAM GCD
C*************************************************************************
C   Program to determine the Greatest Common Divisor
C   of two nonnegative integer values
C*************************************************************************
      IMPLICIT NONE
      INTEGER M, N, TEMP
      PRINT*, 'Type in two nonnegative integers: '
      READ*, M, N
      DO WHILE(N.NE.0)
          TEMP = MOD(M, N)
          M = N
          N = TEMP
      END DO
      PRINT*, 'Their greatest common divisor is', M
      END
```

Sample program output:

```
Type in two nonnegative integers:  48  14
Their greatest common divisor is           2
```

Exercise:

1. What does the following program do? Type and run it. Try different input values for NUMBER.

```
      PROGRAM DIGITS
      IMPLICIT NONE
      INTEGER NUMBER, REVERS, RDIGIT
      PRINT*, 'Enter your number: '
      READ*, NUMBER
```

```
      REVERS = 0
      DO WHILE(NUMBER.NE.0)
          RDIGIT = MOD(NUMBER, 10)
          REVERS = 10*REVERS + RDIGIT
          NUMBER = NUMBER/10
      END DO
      PRINT*, REVERS
      END
```

Example 3.4:

This program illustrates the use of the DO WHILE loop structure to read, count, and analyze a set of exam scores. The use of the library functions NINT, REAL, MIN, and MAX is also exemplified in this program.

```
      PROGRAM SCORES
C*************************************************************************
C   Program to read, count, and analyze exam scores
C   to determine mean, maximum and minimum scores.
C*************************************************************************
      IMPLICIT NONE
      INTEGER COUNT, SCORE, SUM, SMALL, LARGE
C
C   Describe to the user how the scores are to be typed in.
      PRINT*, 'Enter scores as integers between 0 and 100.'
      PRINT*, 'Enter a value outside this range to stop.'
      PRINT*, ' '
C
C   Initialize SUM, COUNT, LARGE and SMALL
      LARGE = 0
      SMALL = 100
      SUM = 0
      COUNT = 0
C
C   Read the first score
      PRINT*, 'Enter the first score: '
      READ*, SCORE
C
C   While not end-of-data, read, sum, and count scores
      DO WHILE(SCORE.GE.0 .AND. SCORE.LE.100)
          COUNT = COUNT + 1
          SUM = SUM + SCORE
          LARGE = MAX(LARGE, SCORE)
          SMALL = MIN(SMALL, SCORE)
          PRINT*, 'Enter score: '
          READ*, SCORE
      END DO
C
C   Display the results
      PRINT*, ' '
      PRINT*, 'Number of scores =', COUNT
      IF(COUNT.GT.0)THEN
          PRINT*, 'Mean score       =', NINT(REAL(SUM)/REAL(COUNT))
          PRINT*, 'Highest score    =', LARGE
          PRINT*, 'Lowest score     =', SMALL
      ENDIF
      END
```

Here is a sample output:

```
Enter scores as integers between 0 and 100.
Enter a value outside this range to stop.

Enter the first score: 44
Enter score: 46
```

```
Enter score: 50
Enter score: 60
Enter score: 58
Enter score: -1

Number of scores =          5
Mean score       =         52
Highest score    =         60
Lowest score     =         44
```

In the calculation of the mean score, the expression

```
NINT(REAL(SUM)/REAL(COUNT))
```

is used to calculate the nearest integer to the actual average score. We see here a typical use of the type conversion function `REAL`. For example, consider the 5 scores `58, 50, 60, 44,` and `46`. `SUM` and `COUNT` will then be `258` and `5`, respectively. The result of the integer division

```
SUM/COUNT
```

would be `51` (fractional part being discarded) whereas the real division

```
REAL(SUM)/REAL(COUNT)
```

yields the value `51.6`. The displayed mean will then be `NINT(51.6)`, i.e. `52`, which is the closest integer to the actual mean score `51.6`. It would actually suffice to convert the value of only one of operands to type real, i.e. we could write

```
REAL(SUM)/COUNT
```

or

```
SUM/REAL(COUNT)
```

as the integer operand of a mixed-mode division is automatically converted to type real before the division is carried out. However, when the intention is the use of real arithmetic, it may be preferable to convert all integers to type real explicitly to avoid any ambiguity.

We also see in this program an application of the functions `MIN` and `MAX`. Note that the statements

```
LARGE = MAX(LARGE, SCORE)
SMALL = MIN(SMALL, SCORE)
```

are equivalent to the statements

```
IF(SCORE.GT.LARGE) LARGE = SCORE
```

and

```
IF(SCORE.LT.SMALL) SMALL = SCORE
```

respectively.

Example 3.5:

In this example, we take up a practically very important problem, namely, *the linear-curve fit problem*. Consider an experiment in which two variables, say X and Y, are measured to establish a relationship between them. In the experiment, N data points, that is (X_i, Y_i) for $i=1$ to N, are obtained. Suppose we next plot the data on a graph paper, and observe that there may be a linear relationship between X and Y. This means that the relationship between X and Y can be described by an equation of the form

$$Y = mX + b$$

where m is the slope of the straight line, and b is the Y-intercept. The problem then becomes one of finding the values of m and b.

Note that, since there are two unknowns (m and b), two measurements (i.e. $N = 2$) would be sufficient to determine them uniquely. This is so because, as a result of two measurements, we obtain two equations

$$Y_1 = mX_1 + b$$
$$Y_2 = mX_2 + b$$

which can be easily solved for m and b. More often than not, however, N is larger than 2, and the problem is *overdetermined*. Since, due to experimental errors or other practical reasons, there will almost always be some scatter in the data, we cannot find any values for m and b that will satisfy all of the equations

$$Y_i = mX_i + b \quad (i=1,...,N)$$

In geometrical terms, no matter how we draw our straight line through the data points, some of the points will not be on the line. The problem may then be stated as follows: find the *best* straight line that represents the given experimental data.

The *Method of Least Squares* solves this problem using the following principle: The straight line should be fitted through the given points [(X_i, Y_i) for $i=1$ to N] so that the sum of the squares of the distances of those points from the straight line is minimum, where the distance is measured in the vertical direction (the Y-direction). The mentioned sum is a measure of error as it is proportional to the discrepancy between the straight line and the data:

$$E = \sum_{i=1}^{N} (Y_i - b - mX_i)^2$$

To find the values of m and b that minimize this error, we set

$$\frac{\partial E}{\partial b} = 0$$

$$\frac{\partial E}{\partial m} = 0$$

After carrying out the resulting algebra, the following equations are found:

$$m = \frac{\sum X_i Y_i - (\sum X_i)\overline{Y}}{\sum X_i^2 - (\sum X_i)\overline{X}}$$

$$b = \overline{Y} - m\overline{X}$$

where

$$\overline{X} = \sum X_i / N \quad \text{and} \quad \overline{Y} = \sum Y_i / N$$

All of the above summations are from $i=1$ to $i=N$. Here is a program that implements these formulas. Notice how the `DO WHILE` loop is used.

```
      PROGRAM LSQUAR
C*************************************************************************
C  Program to find the equation of the least squares line for a set
C  of data points. Variables used are:
C  X, Y     : (X,Y) is an observed data point
C  COUNT    : number of data points
C  SUMX     : sum of X's
```

```
C  SUMY     : sum of Y's
C  SUMX2    : sum of the squares of X's
C  SUMXY    : sum of the products X*Y
C  XMEAN    : mean of the X's
C  YMEAN    : mean of the Y's
C  M        : slope of the line  Y = mX + b
C  B        : Y-intercept of the line Y = mX + b
C*************************************************************************
      IMPLICIT NONE
      INTEGER COUNT
      REAL X, Y, SUMX, SUMY, SUMX2, SUMXY, XMEAN, YMEAN, M, B
C  Initialize the counter and the sums to 0, and read first point
      COUNT = 0
      SUMX = 0.
      SUMXY = 0.
      SUMX2 = 0.
      SUMY = 0.
      PRINT*, 'Enter (X,Y) = (-999, -999) to stop.'
      PRINT*, 'Enter the first point: '
      READ*, X, Y
C  While there are more data, calculate the necessary sums and
C  read the next data point (X,Y).
      DO WHILE((X.NE.-999.).OR.(Y.NE.-999.))
          COUNT = COUNT + 1
          SUMX = SUMX + X
          SUMY = SUMY + Y
          SUMXY = SUMXY + X*Y
          SUMX2 = SUMX2 + X*X
          PRINT*,'Enter the next point X, Y: '
          READ*, X, Y
      END DO
      XMEAN = SUMX/COUNT
      YMEAN = SUMY/COUNT
      M = (SUMXY - SUMX*YMEAN) / (SUMX2 - SUMX*XMEAN)
      B = YMEAN - M*XMEAN
      PRINT*, '  '
      PRINT*, 'Equation of least squares line is Y = mX + b, where '
      PRINT*, 'Slope = m =', M
      PRINT*, 'Y-intercept = b =', B
      END
```

The following is a sample run of this program:

```
Enter (X,Y) = (-999, -999) to stop.
Enter the first point: -1.1  2.05
Enter the next point X, Y: -0.5   3
Enter the next point X, Y: 0    4.1
Enter the next point X, Y: 1.5  6.8
Enter the next point X, Y: 2   8
Enter the next point X, Y: -999 -999

Equation of least squares line is Y = mX + b, where
Slope = m =    1.90956
Y-intercept = b =    4.06437
```

3.4 The DO Loop

The general format of the DO **loop** control structure is the following:

```
    DO n lcv = inv, endv, step
         loop body
n   CONTINUE
```

where *lcv* is the loop-control variable which may be of type real, type double precision, or type integer; the loop parameters *inv, endv,* and *step* denote initial value, end value, and step size (increment), respectively, for the loop-control variable. Properties of the DO loop can be summarized as follows:

1. The loop parameters can be arbitrary integer, real, or double precision arithmetic expressions[2]. (However, *it is recommended that you always use integers as loop parameters. Furthermore, always use an integer variable as the loop-control variable.* See Example 3.9 below.) Each expression is evaluated only once--when the loop is first entered. The values of the variables in these expressions can be modified within the *loop body*, but the values of *inv, endv,* and *step* are not affected by such changes.
2. The *loop body* will be executed once for each value of *lcv*, starting with *lcv* equal to the value of *inv*, and continuing until *lcv* passes the value of *endv*. After each loop execution, the value of *lcv* is automatically updated by the value of *step*.
3. Formally (i.e. according to the FORTRAN 77 standard), when the DO statement is executed an *iteration count* is first calculated using the formula

   ```
   MAX(INT((endv-inv+step)/step),0)
   ```

   and the loop is executed that many times. If the iteration count is zero (e.g. if *inv* > *endv* and *step* > 0), the statements in the *loop body* are not executed at all. In such a case, the loop-control variable *lcv* will still be set to the value *inv*, because this assignment takes place before the iteration count is tested.
4. *step* can be positive or negative but not zero. If *step* is omitted, it is assumed to be +1.
5. Upon exit from the loop, *lcv* will have the value it would have had on the next pass through the loop, if there had been one (see Example 3.7).
6. The value of the loop-control variable *lcv* cannot be altered within the *loop body,* as its value is changed only by the automatic incrementing mechanism after each execution of the *loop body*.

---

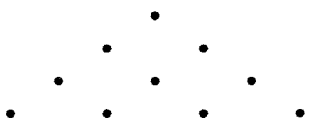
[2] The ability to use real and double precision variables as DO loop indices was added as a new feature in FORTRAN 77. This feature has been deleted from Fortran 95.

7. Transfer of control into the middle of a DO loop from outside by a GO TO statement is not allowed. Control can be transferred from inside *loop body* to outside, however.

You should read these rules again after studying the example programs given below.

Example 3.6:

If one were to display 10 dots into a shape of a triangle, an arrangement that looks like the following would be obtained.



In general, the sum of dots it would take to form a triangle containing N rows would be the sum of integers from 1 to N. This sum is known as a *triangular number*. The following program uses a DO loop to calculate the N-th triangular number, where N is specified by the user.

```
      PROGRAM TRIAN1
C*************************************************************************
C  Calculates the sum 1+ 2+ ... + N using a DO loop
C*************************************************************************
      IMPLICIT NONE
      INTEGER I, N, SUM
      PRINT*, 'What triangular number do you want?'
      READ*, N
      SUM = 0
      DO 1 I = 1,N
          SUM = SUM + I
    1 CONTINUE
      PRINT*,'Triangular number', N, ' is ', SUM
      END
```

Note that SUM is initialized to zero. This is necessary because, SUM must have a value assigned to it (i.e. it must be *defined*) before it is used on the right hand side of the statement SUM=SUM+I during the first execution of the loop.

After the first time the loop body (the statement SUM=SUM+I) is executed, the value of SUM will be 0+1, i.e. 1. After the second iteration, the value of SUM will be 1+2, i.e. 3, and so on.

Consider next the calculation of the N-th triangular number using a DO WHILE loop instead of a DO loop. Note that the loop counter variable I must be initialized before its first use in the logical expression I.LE.N.

```
      PROGRAM TRIAN2
C*************************************************************************
C  Calculates the sum 1+ 2+ ... + N using a DO WHILE loop
C*************************************************************************
      IMPLICIT NONE
      INTEGER I, N, SUM
      PRINT*, 'What triangular number do you want?'
      READ*, N
      SUM = 0
      I   = 1
```

```
      DO WHILE(I.LE.N)
          SUM = SUM + I
          I = I + 1
      END DO
      PRINT*,'Triangular number', N, ' is ', SUM
      END
```

Finally, consider (type and run) the following program that generates the first N triangular numbers and displays them in the form of a table.

```
      PROGRAM TRIAN3
C*************************************************************************
C  Generates a table of triangular numbers
C*************************************************************************
      IMPLICIT NONE
      INTEGER I, N, SUM
      PRINT*, 'Type the largest triangular number in the table:'
      READ*, N
      PRINT*, 'Table of Triangular Numbers'
      PRINT*, '  i    Sum from 1 to i'
      PRINT*, '--------------------------'
      SUM = 0
      DO 1 I = 1,N
          SUM = SUM + I
          PRINT '(1X,I3,8X,I5)', I, SUM
    1 CONTINUE
      END
```

Example 3.7:

```
      PROGRAM LOOP1
      IMPLICIT NONE
C*************************************************************************
C   Program demonstrates some of the properties of the DO loop
C*************************************************************************
      INTEGER I
      PRINT*, 'First loop:'
      DO 1 I = 1, 3
          PRINT*, 'I =', I
    1 CONTINUE
      PRINT*, 'After exit I =', I
C
      PRINT*, ' '
      PRINT*, 'Second loop:'
      DO 2 I = 3, 1, -1
          PRINT*, 'I =', I
    2 CONTINUE
      PRINT*, 'After exit I =', I
C
      PRINT*, ' '
      PRINT*, 'Third loop:'
      DO 3 I = 50, 10, -10
          PRINT*, 'I =', I
    3 CONTINUE
      PRINT*, 'After exit I =', I
      END
```

Note that the same loop-control variable is used in all three loops. This is not necessary, but it is allowed because `I` is reset to the appropriate initial value when each loop is entered. Here is the output of this program:

```
First loop:
I =           1
I =           2
I =           3
After exit I =           4

Second loop:
I =           3
I =           2
I =           1
After exit I =           0

Third loop:
I =          50
I =          40
I =          30
I =          20
I =          10
After exit I =           0
```

Example 3.8:

Consider the problem of reading and processing exam scores (cf. Example 3.4). Suppose that the number of scores is known in advance, and therefore there is no need for the program to count the number of scores. In this example we modify `PROGRAM SCORES` of Example 3.4 to read in the number of scores and use a `DO` loop instead of the `DO WHILE` loop to read in and analyze the scores. Compare the following program with `PROGRAM SCORES` carefully to note all the differences as well as the similarities.

```
      PROGRAM SCORE2
C*************************************************************************
C   Program to read and analyze exam scores
C   to determine mean, maximum and minimum scores.
C*************************************************************************
      IMPLICIT NONE
      INTEGER I, N, SCORE, SUM, SMALL, LARGE
C
C   Read in the number of scores
      PRINT*, 'Enter the number of scores: '
      READ*, N
C
C   Describe to the user how the scores are to be typed in.
      PRINT*, 'Enter scores as integers between 0 and 100.'
      PRINT*, ' '
C
C   Initialize SUM, LARGE and SMALL
      LARGE = 0
      SMALL = 100
      SUM = 0
C
```

```
C   Read and sum N scores
      DO 1 I = 1, N
          PRINT*, 'Enter score: '
          READ*, SCORE
          SUM = SUM + SCORE
          LARGE = MAX(LARGE, SCORE)
          SMALL = MIN(SMALL, SCORE)
    1 CONTINUE
C
C   Display the results
      PRINT*, ' '
      IF(N.GT.0)THEN
          PRINT*, 'Mean score       =', NINT(REAL(SUM)/REAL(N))
          PRINT*, 'Highest score    =', LARGE
          PRINT*, 'Lowest score     =', SMALL
      ENDIF
      END
```

Actually, the last statement of a DO loop does not have to be the CONTINUE statement, but must be an executable statement other than an IF statement, a GO TO statement, a DO WHILE statement, or another DO statement. For example, instead of

```
      DO 1 I = 1, 3
          PRINT*, 'I =', I
    1 CONTINUE
```

we can write

```
      DO 1 I = 1, 3
    1     PRINT*, 'I =', I
```

It is recommended here, however, that you always use the CONTINUE statement as the loop terminator as it serves as a delimiter of the loop and leads to programs that are easier to read.

Example 3.9:

Although FORTRAN 77 allows the use of real expressions as DO loop parameters, you should always use integers as loop parameters. This is because DO loops with real (or double precision) parameters will not always execute the same number of times on different types of computers or with different compilers on the same machine. As a simple illustration, type and run the following program:

```
      PROGRAM BADDO
      IMPLICIT NONE
      REAL X
      DO 1 X = 0.0, 1.0, 0.1
          PRINT '(1X,F10.8)', X
    1 CONTINUE
      END
```

While this program would be expected to print 11 values from `0.0` to `1.0`, it printed only the first 10 values (using either Microsoft PowerStation Version 1.0 on a DOS 6.22/486DX PC or Version 4.0 on a Windows 98/Pentium III machine), the last value printed was `0.90000010` (note that this is not exactly `0.9`). Using Lahey's FORTRAN 77 compiler on the 486DX PC, the program correctly generated 11 values.

Furthermore, the expression (for iteration count) `INT((1.+0.1)/0.1)` evaluated to `11` with Version 4.0 of Microsoft's compiler on the Pentium III PC; it evaluated to `10` with the Version 1.0 compiler running on the 486DX machine. Again, with both systems only 10 values were printed on the screen, the last printed value being `0.90000010`.

You should remember that many decimal fractions cannot be represented exactly in the internal binary format used for real and double precision numbers. The decimal number `0.1` is an example of a real number that cannot be represented exactly by the computer. (The representational error will depend on the number of bits in the mantissa: the more bits, the smaller the error.) Therefore, the result of adding `0.1` ten times may not be equal to `1.0`.

As a solution to the problem we face in the above program, the loop can be rewritten using an integer loop control variable and integer control parameters as follows:

```
      PROGRAM GOODDO
      IMPLICIT NONE
      REAL X
      INTEGER I
      DO 1 I = 1, 11
          X = (I-1)*0.1
          PRINT '(1X,F10.8)', X
    1 CONTINUE
      END
```

Notice how `X` is calculated inside the loop. This loop will always execute exactly 11 times regardless of the processor (computer/compiler combination) used.

3.5 Nested Loops and Block-`IF` Statements

It is possible to have one `DO` loop *lying completely* within the loop body of another `DO` loop. `DO` loops occurring in this way are called *nested* `DO` loops. The following program demonstrates how nested loops are implemented.

Example 3.10:

```
      PROGRAM LOOP2
C*************************************************************************
C   Program to demonstrate how nested DO loops work
C*************************************************************************
      IMPLICIT NONE
      INTEGER I, J, M, N
      PRINT*, 'Enter M and N: '
      READ*, M, N
      PRINT*, '            I           J        I*J'
      PRINT*, '          ---------------------------'
      DO 10 I = 1, N
          DO 20 J = 1, M
              PRINT*, I, J, I*J
   20     CONTINUE
   10 CONTINUE
      END
```

The output corresponding to `M = 3` and `N = 2` is as follows:

```
Enter M and N: 3    2

           I           J        I*J
 ---------------------------------
           1           1          1
           1           2          2
           1           3          3
           2           1          2
           2           2          4
           2           3          6
```

Next consider the following program that prints a set of multiplication tables from `1` times up to `N` times, where each table only goes up to '`I*I`'. Notice the end value (`I`) of the inner loop.

```
      PROGRAM LOOP3
C*************************************************************************
C   Demonstrates how nested DO loops work
C*************************************************************************
      IMPLICIT NONE
      INTEGER I, J, N
      PRINT*, 'Enter N: '
      READ*, N
      PRINT*, '             I           J        I*J'
      PRINT*, '          --------------------------'
      DO 10 I = 1, N
          DO 20 J = 1, I
              PRINT*, I, J, I*J
  20      CONTINUE
          PRINT*, ' '
  10  CONTINUE
      END
```

Example 3.11:

Consider again the calculation of triangular numbers (see Example 3.6). Assume that we need a program that can calculate as many triangular numbers as desired during a single run of the program. Note that the programs `TRIAN1` and `TRIAN2` of Example 3.6 can be used to calculate a single triangular number, whereas `TRIAN3` generates a table of the first `N` triangular numbers, where `N` is specified by the user. The program given below, on the other hand, can be used to calculate any number of triangular numbers in any desired order. Note that there is a `DO` loop lying within the body of a `DO WHILE` loop in this example.

```
      PROGRAM TRIAN4
C*************************************************************************
C  Calculates the N-th triangular number 1+ 2+ ... + N
C  Multiple N values can be requested by the user
C*************************************************************************
      IMPLICIT NONE
      INTEGER I, N, SUM
      CHARACTER*1 ANSWER
      ANSWER = 'Y'
```

```
      DO WHILE(ANSWER.EQ.'Y' .OR. ANSWER.EQ.'y')
           PRINT*, 'What triangular number do you want?'
           READ*, N
           SUM = 0
           DO 1 I = 1,N
               SUM = SUM + I
    1      CONTINUE
           PRINT*,'Triangular number', N, ' is ', SUM
           PRINT*,'Want another calculation? Type Y or y if you do: '
           READ '(A)', ANSWER
      END DO
      END
```

A block-IF structure may also be contained within the body of a DO loop, and a DO loop may be contained in a statement block of a block-IF structure. The following rules should be remembered in dealing with nested control structures: In the case of nested DO loops, the inner loop must lie completely within the outer loop. DO loops and block-IF structures must not overlap. Furthermore, a DO loop contained within a block-IF structure must not overlap two or more tasks of the outer structure. We see that these rules follow from the general rule that was stated in Section 2.6: A*ll nested structures must be wholly contained within a single statement group of the structure(s) they appear in.* In the implementation of nested DO loops the following additional point should be observed: Since the loop-control variable cannot be altered within the loop body, the loop-control variable of the inner DO loop must be different from that of the outer loop. The following are examples of invalid program segments:

```
      DO 10 I = 1, N
            .
            .
         DO 20 J = 1, M
            .
            .
   10   CONTINUE
            .
            .
   20    CONTINUE
```

Unacceptable (overlapping loops)

```
      DO 90 I = 1, N
          .
          .
      IF(I.LT.2)THEN
          .
          .
   90   CONTINUE
          .
          .
      ENDIF
```

Unacceptable (overlapping loop and block-IF)

```
      DO 10 I = 1, N
          DO 20 I = 1, M
              .
              .
   20     CONTINUE
   10 CONTINUE
```

Unacceptable (inner and outer loops use the same loop-control variable `I`)

```
      DO 10 I = 1, N
              .
              .
          I = 2
              .
              .
   10 CONTINUE
```

Unacceptable (loop-control variable `I` cannot be assigned a value within the loop body)

It should be added that nested `DO` loops may have the same last statement. The loop of Example 3.10 can therefore be written as follows:

```
      DO 10 I = 1, N
          DO 10 J = 1, M
              PRINT*, I, J, I*J
   10 CONTINUE
```

Furthermore, since the use of the `CONTINUE` statement is not necessary, the nested loops above could also be written as

```
      DO 10 I = 1, N
          DO 10 J = 1, M
   10         PRINT*, I, J, I*J
```

Again, it is recommended that you include a distinct `CONTINUE` statement for each `DO` loop. Having an unindented loop terminator and an unindented loop header, together with an indented loop body, improves the readability of the program.

Example 3.12:

Given an angle x in radians, the *sine* of *x* has the following infinite series representation:

$$sin(x) = x - x^3/3! + x^5/5! - x^7/7! + x^9/9! - \ldots$$

While theoretically there is an infinite number of terms in this series, a good approximation of the sine of x can be obtained by summing a finite number of the terms. The following program reads in a positive integer `N` and an angle in degrees, converts the angle to radians, and estimates *sin(x)* using the first `N` terms of the series.

```
      PROGRAM SINE
C*************************************************************************
C   Program to calculate the sine of user-entered value X
C
C   Sin(X) = X - X**3/3! + X**5/5! - X**7/7! + ...
C
```

```
C   X    : angle in radians
C   N    : number of terms used in the approximation
C   SUM  : sum of the first N terms
C*********************************************************************
      IMPLICIT NONE
      REAL X, SUM, RADIAN
      INTEGER I, J, N, SGN, FACT
      PRINT*, 'Enter an angle (in degrees) between 0 and 360: '
      READ*, X
C   Convert to radians
      X = RADIAN(X)
      PRINT*, 'Enter the number of terms to be used: '
      READ*, N
C
C   Initialize SUM and SGN
      SUM = 0.0
      SGN = 1
C
C   Sum the first N terms
      DO 1 I = 1, N
          FACT = 1
          DO 2 J = 1, 2*I-1
              FACT = FACT*J
    2     CONTINUE
          SUM = SUM + SGN * X**(2*I-1) / FACT
          SGN = -SGN
    1 CONTINUE
      PRINT*, 'Estimated value of sin(X) =', SUM
      PRINT*, 'Library function sin(X)   =', SIN(X)
      END
C
      REAL FUNCTION RADIAN(THETA)
C*********************************************************************
C   Function to convert degrees to radians
C   THETA: Angle in degrees (input to the function)
C*********************************************************************
      IMPLICIT NONE
      REAL THETA
      REAL PI
      PI = ACOS(-1.)
      RADIAN = PI*THETA/180.
      RETURN
      END
```

The logic of the program is based on the observation that the first term of the series is x, the second is $-x^3/3!$, the third is $+x^5/5!$, etc. so that the i-th term of the series is $x^{(2i-1)}/(2i-1)!$, with the proper sign.

For each value of i, i.e. at each iteration of the outer `DO` loop, the inner `DO` loop computes the factorial $(2i-1)!$.

Note that `SGN` is so initialized that the signs of all the terms are correctly accounted for. Another valid approach would be to eliminate the variable `SGN` completely from the program, and instead use the following expression for summing the series (try it):

```
SUM = SUM + (-1)**(I+1) * X**(2*I-1) / FACT
```

The program also compares the estimated value of the sine of x with the value computed by the library function `SIN`. The following are a couple of sample outputs:

```
Enter angle (in degrees):  60
Enter the number of terms to be used: 5
Estimated value of sin(X) =   0.866025
Library function sin(X)   =   0.866025

Enter angle (in degrees):  90
Enter the number of terms to be used: 4
Estimated value of sin(X) =   0.999843
Library function sin(X)   =    1.00000
```

The above results are excellent. See, however, what happens when we attempt to calculate the sine of 135 degrees:

```
Enter angle (in degrees): 135
Enter the number of terms to be used: 5
Estimated value of sin(X) =   0.707409
Library function sin(X)   =   0.707108
```

We try to get a better estimate of sin(135) by using more terms in the approximation:

```
Enter angle (in degrees): 135
Enter the number of terms to be used: 10
Estimated value of sin(X) =   0.591919
Library function sin(X)   =   0.707108

Enter angle (in degrees): 135
Enter the number of terms to be used: 15
Estimated value of sin(X) =   -56.6007
Library function sin(X)   =   0.707108
```

Note that, with the use of larger numbers of terms, not only the accuracy of the estimate has deteriorated, but the result obtained with `N=15` is simply absurd. Clearly something is amiss.

We go back and review our program to see if there are any programming errors. Obviously, there are no syntax errors in the program. Furthermore, the overall logic of the program appears to be correct as well, and this conjecture is strongly supported by the accurate results obtained for 60 and 90 degrees. Even for 135 degrees, the result is reasonably accurate when 5 terms are used in the approximation, and the problem arises when `N` is large.

We are thus lead to suspect that something goes wrong with the higher order terms, and to pinpoint the exact cause of the problem, we add the following `PRINT` statement for **debugging** the program:

```
          ...
    2     CONTINUE
          PRINT*, 'X**(2*I-1) =', X**(2*I-1), '   FACT=', FACT
          SUM = SUM + SGN * X**(2*I-1) / FACT
```

When we compile and rerun the program with this modification, we get the following output:

```
Enter angle (in degrees): 135
Enter the number of terms to be used: 10
X**(2*I-1)=    2.35619       FACT =           1
X**(2*I-1)=    13.0807       FACT =           6
X**(2*I-1)=    72.6196       FACT =         120
X**(2*I-1)=    403.158       FACT =        5040
X**(2*I-1)=    2238.19       FACT =      362880
X**(2*I-1)=    12425.6       FACT =    39916800
X**(2*I-1)=    68982.7       FACT =  1932053504
X**(2*I-1)=    382967.       FACT =  2004310016
```

```
X**(2*I-1)=   0.212610E+07   FACT =  -288522240
X**(2*I-1)=   0.118033E+08   FACT =   109641728
Estimated value of sin(X) =   0.591919
Library function sin(X)   =   0.707108
```

Noting that x is 2.36 radians, we see that $x^{(2i-1)}$ is accurately calculated for all values of *i* (1 to 10), whereas the factorial of *(2i-1)* is not correct when *i* is greater than 6. Here we identify the cause of the difficulty: The factorial-which is an integer-becomes too large for the computer to handle.

There is a maximum value that an integer constant or variable can have and this maximum value is exceeded in this calculation. (As noted before, the maximum value of an integer usually does not exceed 2,147,483,647, although it varies from computer to computer.) This is termed *integer overflow*.

We can circumvent this difficulty by declaring FACT as a real variable (make sure to try it), but here is a more elegant remedy. The values of $x^{(2i-1)}$ and *(2i-1)!* become individually very large as *i* increases, but the ratio $x^{(2i-1)}/(2i-1)!$ remains small. It is really this ratio that we want to calculate, and not the individual terms $x^{(2i-1)}$ and *(2i-1)!*. In the modified version shown below, the calculation of the factorial *(2i-1)!* and the related variable FACT are eliminated from the program, and instead the inner loop computes the *i*-th term $x^{(2i-1)}/(2i-1)!$ directly and a new variable named TERM is introduced for that purpose.

```
      REAL X, SUM, RADIAN, TERM
      INTEGER I, J, N, SGN
        .....
      DO 1 I = 1, N
          TERM = 1.
          DO 2 J = 1, 2*I-1
              TERM = TERM*X/J
    2     CONTINUE
          SUM = SUM + SGN*TERM
          SGN = -SGN
    1 CONTINUE
```

The following are the results obtained with the new version of our program:

```
Enter angle (in degrees): 135
Enter the number of terms to be used: 10
Estimated value of sin(X) =   0.707108
Library function sin(X)   =   0.707108

Enter angle (in degrees): 135
Enter the number of terms to be used: 15
Estimated value of sin(X) =   0.707108
Library function sin(X)   =   0.707108
```

Exercises:

2. The value of π can be calculated using the following formula:

$$\frac{\pi}{4} = \frac{2\times4\times4\times6\times6\times8\times8\times\ldots}{3\times3\times5\times5\times7\times7\times9\times\ldots}$$

The following program implements this formula:

```
      PROGRAM CALCPI
      IMPLICIT NONE
      INTEGER N, I
      DOUBLE PRECISION FACTOR
      PRINT*, 'Enter N:'
      READ*, N
      FACTOR = 2.D0/3
      DO 1 I = 2, N
          FACTOR = FACTOR * (2*I)/(2*I-1)*(2*I)/(2*I+1)
1     CONTINUE
      PRINT*, 'Pi           = ', 4*FACTOR
      PRINT*, 'ACOS(-1.D0) = ', ACOS(-1.D0)
      END
```

Type and run this program. Try small (e.g. 10) and very large values (e.g. 1,000,000) for `N`. What happens if you declare `FACTOR` as `REAL` ? (Try it.) Also modify the program to write

```
FACTOR = FACTOR * (4*I*I)/(2*I-1)/(2*I+1)
```

instead of the mathematically equivalent form given in the program. What happens when `N` is large and why? Next try the following two versions:

```
FACTOR = FACTOR * 4*I*I/(2*I-1)/(2*I+1)
FACTOR = FACTOR * (4.*I*I)/(2*I-1)/(2*I+1)
```

Hint: Remember the hierarchy of arithmetic operations and the rules used in the evaluation of mixed-mode expressions.

3. The following infinite series converges to *1/2*:

$$1/(1x3) + 1/(3x5) + 1/(5x7) + 1/(7x9) + ...$$

a) Write a main program to calculate and print the sum of the first N terms in the series. The number of terms (N>0) in the summation should be read in by the program. b) Write a main program to estimate the sum of the infinite series as follows. The number of terms used in the summation is not known in advance. The summation will continue until the next term to be added is smaller than a user-entered tolerance `TOL` (a very small positive value). Hint: Use a `DO` loop in part a, use a `DO WHILE` loop in part b.

4. Write a main program to estimate the value of the constant *e* (the base of natural logarithms) using a finite number of terms of the following series:

$$e = 1 + 1/1! + 1/2! + 1/3! + 1/4! + ...$$

The number of terms to be used in the approximation should be read in by the program.

5. Write a program which prints all odd positive integers less than or equal to N (an interactively entered positive number), omitting those integers divisible by 3 or 7.

6. Write a program to calculate and print the following sum:

$$1/1^2 - 1/2^2 + 1/3^2 - 1/4^2 + 1/5^2 - 1/6^2 + ...$$

The number of terms (N) in the summation should be read in by the program.

7. It is known that, if |x| <1, then

$$\frac{\ln(1+x)}{1+x} = x - (1+\frac{1}{2})x^2 + (1+\frac{1}{2}+\frac{1}{3})x^3 - .....$$

Write a function that returns an estimate of *ln(1+x)/(x+1)* using the series given above. The number of terms (N) in the summation and the value of *x* will be input arguments. Write a main program to test this function. Note: The main program should read (or otherwise fix) and check the value of *x* and take an appropriate action depending on that value.

8. Write two main programs to calculate and print an estimate of the following sum:

$$\frac{1}{1^2 2^2 3^2} + \frac{1}{2^2 3^2 4^2} + \frac{1}{3^2 4^2 5^2} + \ldots\ldots$$

a) In the first program, use the first N terms in the series, where N is specified by the user. b) In the second program, continue the summation until the next term to be added is smaller than a user-entered tolerance `TOL` (a very small positive value). Hint: Use a `DO` loop in part a, use a `DO WHILE` loop in part b.

3.6 Structured Programming and Standard Control Structures

Undisciplined use of certain programming language constructs, most notably the `GO TO` statement, can lead to programs which are very difficult to read. The large cost of maintaining and modifying large programs after they are placed into service requires that programs be written to be easily understood, modified, and further developed by programmers other than the original authors. Largely in response to this requirement, certain principles of good programming practice have evolved over the years, and this has given rise to the concept of **structured programming**.

The programming activity referred to as **maintenance** involves fixing bugs and modifying programs to handle new and unforeseen situations. **Maintainability** is the ease with which a program can be maintained after it is written and tested. The need for such maintenance may arise either from undetected errors in the program or from changes to the original requirements. It is quite common for more time and resources to be spent in maintaining a program than was spent in originally developing it. In general, structured programming leads to programs that are easier to maintain and therefore helps reduce maintenance costs.

While the term 'structured programming' means many different things to many different people, structured programming can be defined as a style of programming designed to make programs more comprehensible. Structured programs are more readable (compared to "unstructured programs") and can be more easily understood and modified. For example, of the following two program segments, the first is said to be structured, while the second is not:

Structured:

```
      IF(NUMBER.LT.0)THEN
          SGN = -1
      ELSE IF(NUMBER.EQ.0)THEN
          SGN = 0
      ELSE
          SGN = 1
      ENDIF
      PRINT*, SGN
```

Unstructured:

```
        IF(NUMBER.EQ.0) GO TO 1
        IF(NUMBER.GT.0) GO TO 2
        SGN = -1
        GO TO 3
    1   SGN = 0
        GO TO 3
    2   SGN = 1
    3   PRINT*, SGN
```

Notice that, while the two program segments above accomplish exactly the same task, it is much easier to tell what the first segment does.

In Section 3.1, we have defined *control structures* as program structures/statements which affect the order in which statements are executed, or which affect whether statements are executed at all. These are things like DO-loops, block-IF's, DO WHILE loops, and so on. The goals of structured programming can be accomplished in two ways. First, the essential control structures which are needed repeatedly in programming must be learned and well-understood. Second, the use of control statements whose controlled blocks are difficult to distinguish at a glance should be avoided as much as possible. A consequence of this principle is that, care should be exercised when using the GO TO control statement in FORTRAN programs.

The above remark about the GO TO statement warrants further explanation. Execution of the GO TO statement causes a direct transfer of control to a specified point in the program. This transfer is affected immediately upon the execution of the GO TO. In order to identify to where in the program the transfer is to be made, a *label* (statement number) is used. It is actually not the GO TO's which are potentially harmful (although they may interrupt the normal sequential flow and hence one's reading of a program); the statement labels are the real hazard: Whenever one encounters a label while reading a program, the questions that immediately come to mind are "Where does control come *from* to this statement?" (It could come from

anywhere in the program) and "What is the condition that results in the transfer of control to this statement?" (It could be anything!). In reading a program that includes a lot of GO TO statements and labeled statements, the reader can easily become confused as to the exact conditions under which a particular group of statements will be executed. Such programs are sometimes referred to as *spaghetti programs* because of their heavily intertwined logical structure.

It should be added that essentially every programming language contains the GOTO statement. Thus we read on p.359 of *Programming in C* by Kochan (1990): "The goto statement interrupts the normal sequential flow of a program. As a result, programs are harder to follow. Using many gotos in a program can make it impossible to decipher. For this reason, goto statements are not considered part of a good programming style."

A similar point of view is expressed by Gottfried (1985) on p.112 of *Programming with Pascal*: "...it should be recognized that the use of the GOTO statement is discouraged in Pascal, since it alters the clear, sequential flow of logic that is characteristic of the language. In fact some computer scientists advocate a total ban on the GOTO statement, though it may be a bit extreme. There is widespread agreement, however, that the GOTO statement should be used very sparingly, and only when it is awkward to use another control structure. Actually, such situations are quite rare."

There may be cases, however, in which a program segment written with the GO TO is easier to understand than one without it. In other cases, e.g. when the DO WHILE structure is not available on a particular compiler or to implement the Do-Until and the Break iterations (see below), the GO TO statement becomes useful and simply has to be used. On the other hand, if a particular type of control structure can be implemented easily by using constructs such as block-IF, the DO loop, or the DO WHILE loop (or the DO...END DO structure of Fortran 90), then these may be employed and the use of the GO TO statement can be avoided.

It should be added that structured programming is more a popular movement than a precise theory. While the above discussion may give the impression that structured programming is synonymous with either "GOTO-less programming" or "minimal use of the GOTO statement," this is not exactly true for at least two reasons.

First, the actual distinguishing feature of a structured program (as opposed to an unstructured program) is that it is easily readable and comprehensible. There are indeed programmers who make liberal use of the GO TO statement and still write

readable and high quality code. It is also possible to write low quality and difficult to read programs without using any GO TO statements or statement labels. Thus, "GOTO-less programming" is neither a necessary nor a sufficient condition of structured programming. One point on which everybody agrees, however, is that careless and undisciplined used of GO TO statements may quickly lead to highly unreadable programs. The main lesson here is, *you should adopt and use programming practices and a programming style that lead to well-organized, readable, and maintainable programs.*

Second, in addition to avoiding undisciplined use of GO TO statements, there are other very important programming practices and principles that contribute to the legibility and maintainability of programs. Inserting comments at appropriate places in a program, using meaningful program and variable names, indenting blocks of code within control structures, using extra blanks at appropriate places to improve readability, etc. are examples. **Modularity** is also an important feature of structured programs. It is strongly recommended that you always break your programs into small subprograms (functions or subroutines), each of which has a logically single and coherent purpose. A commonly recommended practice is to keep program units to no more than 50 lines of code. In this way, each program unit can be printed on a single sheet of paper and more easily examined.

Structured programming, furthermore, is not the only important consideration in the development of high-quality software; **efficiency** and **portability** are also very important. Thus we read in *Efficient Fortran Programming* by Kruger (1990): "Computing efficiency does not evoke quite the feelings that the GO TO statement does, but it is notwithstanding a contentious issue. It surely does not hold the position it used to in the 1960s and early 1970s. At one point, some authors viewed the whole matter of efficiency almost with disgust, placing all emphasis on program structure and maintainability. Unfortunately this approach will often lead to rather inefficient programs, and most professional programmers and computer scientists now realize that there is more to programming than structure."

We have indicated that, to be able to develop structured programs, one has to have a good knowledge of certain essential control structures that are needed over and over again in programming. These control structures are the following:

Simple Iteration: In FORTRAN, simple iteration is performed with a DO loop. The DO loop (see Section 3.4) has the following general format:

```
DO n lcv = inv, endv, step
      loop body
n CONTINUE
```

The majority of modern FORTRAN 77 compilers provide a nonstandard language extension, namely the END DO **statement**[3], that obviates the statement label *n*. Eliminating statement labels in this way may improve the readability of programs. This statement is employed as follows:

```
DO lcv = inv, endv, step
      loop body
END DO
```

IF Structure: The block-IF structure and its special cases, the single-alternative and the double-alternative forms, were discussed in Chapter 2.

Do-While Iteration: The DO WHILE loop described in Section 3.3 implements the Do-While iteration. As explained in that section, the DO WHILE loop is not part of standard FORTRAN 77, but it is included by many modern FORTRAN 77 compilers as a non-standard feature:

```
DO WHILE(logical expression)
      loop body
END DO
```

In anticipation of the discussion of the *Do-Until iteration* below, the following point should be emphasized: The loop repetition test (i.e. evaluation of *logical expression*) for the DO WHILE loop is at the beginning of the loop. The *loop body* will not be executed at all if the repeat condition (*logical expression*) is false the first time that the repetition test is encountered.

Do-Until Iteration: The Do-Until iteration is another method of looping that is occasionally needed in programming. Standard FORTRAN 77 does not provide a structure (like the REPEAT-UNTIL loop in Pascal) that carries out this type of iteration. The GO TO statement, however, can easily be employed to implement the Do-Until iteration. Here is the general format:

```
n CONTINUE
      loop body
IF(logical expression) GO TO n
```

[3] The END DO statement has been included in standard Fortran 90.

The *loop body* is repeated until *logical expression* is false. This type of iteration is similar to the Do-While iteration (i.e. the DO WHILE loop) except for one important difference: the loop repetition test is at the end of the loop. This means that the *loop body* will always be executed at least once, i.e. even if the repeat condition (*logical expression*) is false the first time that the repetition test is encountered.

Break Iteration: In this type of iteration, the *loop body* is repeated until some condition *tested at one or more points in the middle of the loop* becomes true. The following is how the Break iteration can be implemented in FORTRAN:

```
m CONTINUE
        statements before the termination test
        IF(logical expression) GO TO n
        statements after the termination test
    GO TO m
n CONTINUE
```

Iterations using the DO Construct: All of the mentioned types of iteration can be implemented using a single control structure, namely the **DO...END DO construct,** in Fortran 90. It therefore seems appropriate to introduce this new Fortran 90 feature at this stage. Note that this construct is available as a nonstandard extension in many commercial FORTRAN 77 compilers.

The DO...END DO structure takes one of two possible forms. The first form (already mentioned above) is similar to the DO loop of standard FORTRAN 77 and is referred to as the **count-controlled DO loop**:

```
DO lcv = inv, endv, step
    loop body
END DO
```

Note that this is simple iteration. The other possible form of the DO...END DO structure is

```
DO
    loop body
END DO
```

In this case, there is no loop control variable *lcv*. Consequently, there must be some other means of stopping the loop. This can be done by using the Fortran 90 **EXIT statement**, also available as a nonstandard extension in many FORTRAN 77 compilers. The EXIT statement inside a loop causes the loop to terminate and

execution continues with the first executable statement following the END DO statement. The Do-While iteration can therefore be implemented as follows.

```
DO
IF(.NOT.logical expression) EXIT
    loop body
END DO
```

The Do-Until iteration is also easily implemented:

```
DO
    loop body
IF(.NOT.logical expression) EXIT
END DO
```

Here is how Break iteration is implemented:

```
DO
  statements before the termination test
  IF(logical expression) EXIT
  statements after the termination test
END DO
```

3.7 Obsolescent Control Structures of FORTRAN

It was noted earlier (see Section 2.5) that the block-IF construct was not available before FORTRAN 77. As a matter of fact, the logical IF statement was the most powerful decision-making statement available at that time. As a result, programmers employed the GO TO statement in conjunction with the logical IF statement to construct control structures which can now be implemented using the block-IF structure. This is illustrated in the next example.

Example 3.13:

Since there is no way to include more than one dependent statement in a single logical IF statement, FORTRAN IV/66 programmers used to employ the GO TO statement to handle multiple dependent statements. Consider the following FORTRAN 77 program that uses this older approach.

```
PROGRAM OLDIF
IMPLICIT NONE
REAL X, SUMNEG, SUMPOS
INTEGER NNEG, NPOS
SUMNEG = 0.
SUMPOS = 0.
NNEG = 0
NPOS = 0
PRINT*, 'Program to count and sum positive and negative'
PRINT*, 'values. Type 0 to mark the end of your data.'
PRINT*, 'Enter first value: '
READ*, X
```

```
C   Begin summation and counting loop.
    1 IF(X.EQ.0) GO TO 10
          IF(X.GT.0) GO TO 2
          NNEG = NNEG + 1
          SUMNEG = SUMNEG + X
          GO TO 3
    2     NPOS = NPOS + 1
          SUMPOS = SUMPOS + X
    3     PRINT*, 'Enter next value: '
          READ*, X
      GO TO 1
   10 CONTINUE
C   Loop ended. Print results.
      PRINT*, 'Results: '
      PRINT*, 'Number of neg values =', NNEG
      PRINT*, 'Number of pos values =', NPOS
      PRINT*, 'Sum of neg values =', SUMNEG
      PRINT*, 'Sum of pos values =', SUMPOS
      END
```

The program given above is certainly a valid FORTRAN 77 (and, Fortran 90) program. However, it is possible to rewrite the program in a more readable form using the block-`IF` construct and a `DO WHILE` loop:

```
      DO WHILE(X.NE.0)
          IF(X.LT.0)THEN
              NNEG = NNEG + 1
              SUMNEG = SUMNEG + X
          ELSE
              NPOS = NPOS + 1
              SUMPOS = SUMPOS + X
          ENDIF
          PRINT*, 'Enter next value: '
          READ*, X
      END DO
```

There are three other obsolete decision-making constructs in FORTRAN that needs to be mentioned. These are (i) the arithmetic `IF` statement, (ii) the computed `GO TO` statement, and (iii) the assigned `GO TO` statement. *All of these control statements can lead to very hard to understand and error-prone programs.* Therefore their use in new programs is strongly discouraged. The arithmetic `IF` statement and the computed `GO TO` statement have been declared obsolescent in Fortran 95, which means they are candidates for deletion from the language in its future versions. The `ASSIGN` statement and the assigned `GO TO` statement were declared obsolescent in Fortran 90 and have been removed from Fortran 95, which means they are no longer a part of Fortran. Commercial compilers, however, may be expected to continue to support these statements as nonstandard features.

It should be noted that these statements had been used extensively by FORTRAN programmers in the 1960s and the 70s (and perhaps by some in the 80s), and you may come upon code that contains one or more of these old constructs. The applicable rules are therefore briefly explained and a few simple examples are given in what follows.

Arithmetic IF *Statement*: The general form of the **arithmetic IF statement** is

IF(*arithmetic expression*) $Label_1$, $Label_2$, $Label_3$

where $Label_1$, $Label_2$, $Label_3$ are the labels (statement numbers) of executable statements in the same program unit. The same statement label may appear more than once. The *arithmetic expression* is an ordinary arithmetic expression of type integer, real, or double precision. This expression is first evaluated. If it is negative, then control is transferred to the statement referenced with $Label_1$. If the expression is zero, control passes to statement labeled $Label_2$. If it is positive, control passes to the statement referenced with $Label_3$. Thus, the arithmetic IF statement has the same effect as the following three consecutive statements:

IF(*arithmetic expression*.LT.0)GO TO $Label_1$
IF(*arithmetic expression*.EQ.0)GO TO $Label_2$
GO TO $Label_3$

The program of Example 3.13 could have been written as follows:

```
 1 IF(X) 4, 10, 2
 4      NNEG = NNEG + 1
        SUMNEG = SUMNEG + X
        GO TO 3
 2      NPOS = NPOS + 1
        SUMPOS = SUMPOS + X
 3      PRINT*, 'Enter next value: '
        READ*, X
   GO TO 1
10 CONTINUE
```

Computed GO TO *Statement*: The general form of the **computed** GO TO **statement** is

GO TO($Label_1$, $Label_2$, $Label_3$, ..., $Label_m$) *integer expression*

where $Label_1$, $Label_2$ etc. are the labels of *m* executable statements in the same program unit. The number of labels *m* in the list is 1 or greater. The same statement label may appear more than once in this list. If the value of *integer expression* is 1, then control is transferred to $Label_1$. If the value is 2, control passes to $Label_2$, and so on. If the value of *integer expression* is out of range, i.e. if it is less than 1 or greater than *m*, then the statement following the computed GO TO statement is executed.

Example 3.14:

The use of the computed the GO TO statement is exemplified in the following program. The variable MONTH may have valid values between 1 (for January) and 12 (for December). Note that winter includes the first two (1st and 2nd) and the last (12th) months of the year.

```
      PROGRAM COMPGO
C*************************************************************************
C   Program determines the season of the year
C   for a given month number between 1 and 12.
C*************************************************************************
      IMPLICIT NONE
      INTEGER MONTH
      PRINT*, 'Enter month number: '
      READ*, MONTH
      GO TO (1,1,2,2,2,3,3,3,4,4,4,1) MONTH
          PRINT*, MONTH, ' is not a valid month.'
          STOP
    1     PRINT*,'Season is winter.'
          GO TO 20
    2     PRINT*,'Season is spring.'
          GO TO 20
    3     PRINT*,'Season is summer.'
          GO TO 20
    4     PRINT*,'Season is autumn.'
   20 CONTINUE
      PRINT*, 'End Computed GO TO Illustration'
      END
```

Assigned GO TO *Statement*: To explain the assigned GO TO statement, it is first necessary to discuss the **ASSIGN statement**. This statement assigns the value of a statement or format label to an integer variable. The general form is as follows:

ASSIGN *label* TO *integer_variable*

where *label* is an integer constant representing a format label or a statement label. It should be emphasized that the effect of the ASSIGN statement is different from that of a computational assignment statement (which uses the = operator). For example, the numerical value printed for the integer variable N after the execution of

```
ASSIGN 100 TO N
PRINT*, 'N =', N
```

will not be 100 (the value printed is compiler-dependent). Assuming that N will have the value 100 after the execution of the above ASSIGN statement would lead to erroneous results. After the ASSIGN statement has been executed, therefore, *integer_variable* should not be used in an arithmetic expression; it should be used only in an assigned GO TO statement or as the label of a FORMAT statement within

the same program unit. This subtle point may be better appreciated after studying Example 3.15 (see below).

The **assigned GO TO statement** has the following general form:

GO TO *integer_variable*, ($Label_1$, $Label_2$, $Label_3$, ..., $Label_m$)

The current label assigned to *integer_variable*, at the time such a statement is executed, must be one of the labels $Label_1$, $Label_2$, $Label_3$, ..., $Label_m$. Otherwise, an execution-time error occurs. The labels in the list should belong to other executable statements within the same program unit. The assigned GO TO statement transfers control to the statement whose label is currently assigned to *integer_variable*. The statement label assigned to *integer_variable* can be changed by another ASSIGN statement in the same program unit. These points and the use of ASSIGN to specify a FORMAT statement label are illustrated in the next example.

Example 3.15:

To understand how the ASSIGN statement is employed, type and run the following program. Note that X and N are ordinary real and integer variables, respectively. On the other hand, the integer variable M is first employed as a FORMAT statement label. It is later used in the same program as the target label of an assigned GO TO statement.

```
      PROGRAM ASSGGO
C*************************************************************************
C   Program demonstrates the ASSIGN statement
C   and the assigned GO TO statement.
C*************************************************************************
      IMPLICIT NONE
      REAL X
      INTEGER N, M
C   Employ M as a FORMAT statement label
      X = ACOS(-1.)
      N = 1
      ASSIGN 20 TO M
      PRINT M, N
      ASSIGN 30 TO M
      PRINT M, X
C   Use M as the target label of an assigned GO TO statement
      ASSIGN 200 TO M
      GO TO M, (100, 200)
  100 PRINT*, 'Statement labeled 100. This will not be printed.'
  200 PRINT*, 'Statement labeled 200. This must be printed.'
      PRINT*, 'End ASSIGN Statement Illustration'
C   FORMAT statements
   20 FORMAT(1X, 'Using FORMAT 20.      N =', I3)
   30 FORMAT(1X, 'This is FORMAT 30.    X =', F9.5)
      END
```

Next, modify the program so that it contains the following segments:

```
      M = -99
      PRINT*, 'M =', M
      ASSIGN 20 TO M
      PRINT*, 'After assigning 20 to M, M =', M
       ...
      PRINT*, 'End ASSIGN Statement Illustration'
      M = 99
      PRINT*, 'M =', M, ' 2*M =', 2*M
```

Once all the I/O and the assigned GO TO statements employing it are executed, *integer_variable* may be assigned a value using a computational assignment statement or a READ statement. It may then appear in an arithmetic expression as an ordinary integer variable. The second program segment above exemplifies this point. Between the execution of the ASSIGN statement and the execution of the assigned GO TO or the I/O statements that employ *integer_variable*, however, a computational assignment statement or a READ statement should not be used to define *integer_variable*. To observe this restriction, carry out the following modification and run the program again:

```
ASSIGN 30 TO M
M = 55
PRINT M, X
```

3.8 Arrays

In all the programs we have studied so far, a distinct name was used for each variable value. In certain applications, however, it is preferable to employ the same name to refer to a list of values that have the same common characteristics. This is done by the use of **arrays** or subscripted variables.

When we carry out mathematical operations with a pen on a piece of paper (or maybe with a computer and using word-processing software that can handle mathematical symbols), we use a symbol with a varying subscript to denote the elements of a given set. For example, T_1, T_2,...,T_{10} may be used to represent 10 distinct temperature measurements. With the utilization of subscripts, we can develop very concise notation for a variety of mathematical operations on these values. Thus, to express mean temperature we can write

$$\frac{1}{10}\sum_{i=1}^{10} T_i$$

instead of

$$(T_1 + T_2 + T_3 + T_4 + T_5 + T_6 + T_7 + T_8 + T_9 + T_{10})/10$$

Subscripts are also used in FORTRAN, although they have a slightly different appearance: we write T(1), T(2), ...,T(10) instead of T_1, T_2,...,T_{10}. A symbolic name that is used with subscripts--such as T here--is called an *array* in FORTRAN.

To further demonstrate the motivation for the use of arrays, consider the problem of analyzing 10 temperature measurements. Suppose that the analysis to be carried out requires all 10 values to be saved and stored, as opposed each temperature value being "forgotten" after being read in and used once. To store the 10 values, we need to introduce 10 variable names, say T1, T2,..., T10. (We can of course use variable names like A, BCV, XE9Y, C, etc., but clearly T1, T2,... are more meaningful names for temperature values.) To calculate the mean temperature we can then use the following program statement:

```
MEANT=(T1+T2+T3+T4+T5+T6+T7+T8+T9+T10)/10
```

We can easily write a statement like this. Now imagine that we have 10,000 (ten thousand) temperature measurements. To write the necessary FORTRAN statement would require an dauntingly large amount of keypunching. We could do it, but such an exercise is certainly not conducive to mental well-being. The solution of this problem is, however, greatly simplified by defining an array of variables. To that end, we write

```
REAL T(10000)
```

at the top of our program, before any executable statement. This array declaration instructs the compiler to associate 10000 memory cells, designated as `T(1)`, `T(2)`, ..., `T(9999)`, and `T(10000)` with the name `T`. We can use this array to store the 10,000 temperature measurements and then easily compute the mean temperature by including a set of statements such as the following in our program:

```
      SUM = 0.0
      DO 1 I = 1, 10000
          SUM = SUM + T(I)
1     CONTINUE
      MEANT = SUM/10000.
```

A linear array declaration, in the most general form, has the following format:

type name(low:high)

where *type* may be any of the FORTRAN data types, i.e. `INTEGER`, `REAL`, `CHARACTER`, etc.; *low* denotes the smallest subscript, and *high* denotes the largest allowed subscript for the array indicated by *name*. The individual array elements are written as *name(low)*, *name(low+1)*, ..., *name(high)*. Clearly, *low* has to be less than or equal to *high*, and the number of elements of the array is *(high - low + 1)*. The following are examples of valid array declarations:

```
REAL T(1:10000)
INTEGER AGE(0:49)
CHARACTER *24 NAME(1:100)
INTEGER COUNT(-5:12)
```

There is a simpler format that is frequently used in array declarations:

type name(size)

where *size* has to be a positive integer. When an array is declared in this way, the smallest subscript is automatically assumed to be `1`. The largest allowable subscript

is then equal to *size*. For example, the following two declarations are exactly equivalent:

```
REAL T(1:10000)
REAL T(10000)
```

The DIMENSION statement can also be used to declare an array. This statement has the following form:

DIMENSION *name*(*low:high*)

The DIMENSION statement, however, does not declare the type of the array. Consequently, a separate type declaration is needed to specify the type of the array. For example, to declare INDEKS as a real array with 200 elements, we would have to use the following two lines of code

```
REAL INDEKS
DIMENSION INDEKS(200)
```

instead of the single line

```
REAL INDEKS(200)
```

In defining and using arrays, the following points should be remembered: One or more arrays and simple variables can be declared in the same declaration statement provided that commas are used between entries in the declaration. Array elements may be manipulated just as other variables are manipulated in program statements. An array element is referenced by writing the name of the array followed by a pair of parentheses enclosing a subscript. The subscript can be an integer constant, an integer variable, or an integer arithmetic expression whose value is not less than *low* and not greater than *high*. If the subscript falls outside the declared range, then a run-time error which may manifest itself in very unpredictable ways will result. It is therefore very important to make certain that all subscripts remain within their allowed ranges.

Example 3.16:

```
      PROGRAM ARRAYS
C*************************************************************************
C   Program demonstrates how linear arrays are declared and used
C*************************************************************************
      IMPLICIT NONE
      INTEGER I, ORDER(-2:3), DOUBLE(4)
C
C   Assign values to all array elements
```

```
      DO 1 I = 1, 4
          DOUBLE(I) = 2*I
    1 CONTINUE
      DO 2 I = -2, 3
          ORDER(I) = I + 3
    2 CONTINUE
C
C   Print the values
      PRINT*, 'Elements of the array ORDER: '
      DO 3 I = 1, 6
          PRINT*, ORDER(I-3)
    3 CONTINUE
      PRINT*, ' '
      PRINT*, 'Elements of the array DOUBLE: '
      DO 4 I = 1, 4
          PRINT*, DOUBLE(I)
    4 CONTINUE
      END
```

Note that, for printing the elements of ORDER, we could have used the alternative loop

```
      DO 3 I = -2, 3
          PRINT*, ORDER(I)
    3 CONTINUE
```

The execution output of the program is the following:

```
Elements of the array ORDER:
           1
           2
           3
           4
           5
           6

Elements of the array DOUBLE:
           2
           4
           6
           8
```

Example 3.17:

The *Fibonacci* sequence of numbers f_0, f_1, f_2, ... is defined $f_0=0$, $f_1=1$, and the general formula $f_{i+2} = f_{i+1} + f_i$ (where $i=0,1,2,...$). This sequence is directly related to the "rabbits problem."

Consider starting with a pair of grown rabbits. Assume the following: 1) Each pair of rabbits produces a new pair of rabbits each month. 2) Each newly born pair can produce offspring by the end of their second month. 3) Rabbits do not die in the time period considered. The rabbit problem can now be stated as follows: How many pairs of rabbits will there be at the end of *i*th month? A few minutes' consideration will show that the answer is f_{i+2}.

```
      PROGRAM RABBIT
C*************************************************************************
C   Calculates and prints the first 20 Fibonacci numbers
C*************************************************************************
      IMPLICIT NONE
      INTEGER F(0:19), I
```

```
C  F0 and F1 are fixed by the following definitions
       F(0) = 0
       F(1) = 1
C  Calculate subsequent Fibonacci numbers
       DO 1 I = 2, 19
           F(I) = F(I-2) + F(I-1)
     1 CONTINUE
C  Print the numbers
       DO 2 I = 0, 19
           PRINT*, F(I)
     2 CONTINUE
       END
```

Example 3.18:

Press et al., in their acclaimed book named *Numerical Recipes*, state[4] "If you know what bubble sort is, wipe it from your mind; if you don't know, make a point of never finding out!" Well, it is now our turn to find out what bubble sort is.

Bubble-sort method is an algorithm for arranging a group of N numbers in ascending order. The method works like the following.

First, we compare the first two numbers in the list. If they are in order, we leave them alone; otherwise, we interchange them. We do the same thing with the second and the third numbers, then the third and the fourth, until we reach the last two numbers. At this point we are guaranteed that the largest number in the list has "bubbled" up the list so that it is now the last number in the list.

Next, we repeat the entire process for the first N-1 numbers. After we do this, the second largest number will be in the second to last place. We keep repeating this process, and after N-1 sweeps, the list of numbers will be sorted in increasing order. The following is a program that implements this method:

```
      PROGRAM BUBBLE
C*************************************************************************
C   Program reads in an array of integers in arbitrary order,
C   and rearranges and prints the array in ascending order.
C*************************************************************************
      IMPLICIT NONE
      INTEGER I, J, N, TEMP, VALUE(100)
C
C   Read input
      PRINT*, 'Enter the number of values: '
      READ*, N
      IF(N.GT.100) THEN
          PRINT*, 'N is too large. Execution terminated.'
          STOP
      ENDIF
      DO 1 I = 1, N
          PRINT*, 'Enter value: '
          READ*, VALUE(I)
    1 CONTINUE
```

[4] Numerical Recipes, by the way, is a very useful reference on numerical methods and contains a large number of widely used FORTRAN programs.

```
C
C   Bubble sort
      DO 2 I = 1, N-1
          DO 3 J = 1, N-I
              IF(VALUE(J).GT.VALUE(J+1))THEN
                  TEMP = VALUE(J+1)
                  VALUE(J+1) = VALUE(J)
                  VALUE(J) = TEMP
              ENDIF
    3     CONTINUE
    2 CONTINUE
C
C   Print the sorted array
      PRINT*, ' '
      DO 4 I = 1, N
          PRINT*, VALUE(I)
    4 CONTINUE
      END
```

Shown below is a sample run:

```
Enter the number of values: 5
Enter value: 9
Enter value: -3
Enter value: 4
Enter value: -5
Enter value: 5

          -5
          -3
           4
           5
           9
```

3.9 Generating Prime Numbers

Just about anyone learning to program is soon faced with the problem of writing (or reading and understanding) a program that produces a table of prime numbers. This is not because such a table has a notable use in engineering and science, but is rather due to the fact that by considering this problem, it is possible to explain many significant programming concepts and to exemplify several different constructs of the particular programming language that is being studied. We take up this problem here to illustrate and hopefully to further clarify various features of FORTRAN we have discussed so far. These features include, among others, the `DO` loop, the `DO WHILE` loop, the use of nested control structures, the logical `IF` statement, the `GO TO` statement, the use logical variables, and linear arrays. We shall also consider the matter of *efficiency*.

A positive integer is a prime number if it is not evenly divisible by any other integer, other than 1 and itself. The first prime integer is defined to be 2. The next

prime is 3, as it is not evenly divisible by any integer other than 1 and 3. The integer 4 is not a prime number because it is evenly divisible by 2. There are several methods that can be used to generate a table of prime numbers. If we want to generate all prime numbers up to `N`, then the simplest approach to accomplish this is to test each integer `I` up to `N` for divisibility by all integers from 2 to `I-1`. If any such integer evenly divides `I`, the `I` is not a prime; otherwise it is a prime. The following program implements this approach.

```
      PROGRAM PRIME1
C*************************************************************************
C   Determines and displays prime numbers up to user-entered value N
C*************************************************************************
      IMPLICIT NONE
      LOGICAL PRIME
      INTEGER I, J, N
      PRINT*, 'Enter N: '
      READ*, N
      PRINT*, 'Prime numbers up to', N, ':'
      PRINT*, ' '
C
C   Test integer I for divisibility by all integers from 2 thru I-1:
C   I is prime if it is not divisible by any such integer.
      DO 1 I = 2, N
          PRIME = .TRUE.
          DO 2 J = 2, I-1
              IF(MOD(I, J) .EQ. 0) PRIME = .FALSE.
    2     CONTINUE
          IF(PRIME) PRINT*, I
    1 CONTINUE
      END
```

Here is a sample run of this program:

```
Enter N: 60
Prime numbers up to          60:

           2
           3
           5
           7
          11
          13
          17
          19
          23
          29
          31
          37
          41
          43
          47
          53
          59
```

Note that the outermost DO loop cycles through the integers 2 through N. The loop variable I represents the value we are currently testing to see if it is prime. The first statement in the loop assigns the value .TRUE. to the logical variable PRIME. The inner loop is set up to divide I by integers from 2 to I-1. Inside the loop a test is made to see if the remainder of the division I/J is zero. If it is, then I is not prime because there exists an integer other than 1 and I itself that evenly divides I. In that case, PRIME is set equal to .FALSE. to signal that I is no longer a candidate as a prime number. When the inner loop finishes execution, PRIME is tested to see if it is true. If it is, then no integer was found that evenly divided I. Hence, I is prime and its value is displayed. If PRIME is not true at the end of the execution of the inner loop, then there was at least one integer between 2 and I-1 that evenly divided I. In that case, I is not prime and therefore its value is not printed.

While the program above generates the desired table, it has a number of inefficiencies. The most obvious inefficiency results from checking even numbers. Clearly, all even numbers greater than 2 are divisible by 2, and therefore cannot be prime. The program should therefore skip even numbers as possible primes *and* as possible divisors. (Remember that if an integer is evenly divisible by an even integer, then it is also evenly divisible by 2, and hence cannot be odd. Since odd numbers are not evenly divisible by even numbers, there is no need to consider even numbers as divisors.) The inner loop of the program is also inefficient because the value of I is *always* tested for divisibility by *all* values of J from 2 to I-1. Actually, there is no need to continue this testing once a number that divides I evenly is found, i.e. once PRIME has been assigned the value .FALSE.. This inefficiency will be removed if the execution of the inner loop is terminated as soon as PRIME is false. This can be done by employing a DO WHILE loop (instead of a DO loop) as the inner loop. The following program implements these improvements:

```
      PROGRAM PRIME2
C*************************************************************************
C   Determines and displays prime numbers up to user-entered value N
C*************************************************************************
      IMPLICIT NONE
      LOGICAL PRIME
      INTEGER I, J, N
      PRINT*, 'Enter N: '
      READ*, N
      PRINT*, 'Prime numbers up to', N, ':'
      PRINT*, ' '
      IF(N.GE.2) PRINT*, 2
C
C   Test odd integer I for divisibility by all odd integers from 3 thru I-2:
C   Odd integer I is prime iff it is not divisible by any such integer.
```

```
      DO 1 I = 3, N, 2
          PRIME = .TRUE.
          J = 3
          DO WHILE(PRIME .AND. J.LE.I-2)
              IF(MOD(I,J) .EQ. 0) PRIME = .FALSE.
              J = J + 2
          END DO
          IF(PRIME) PRINT*, I
    1 CONTINUE
      END
```

While PROGRAM PRIME2 is more efficient than PROGRAM PRIME1, it is still not very efficient. It is true that the issue of efficiency is not very important when we want to generate a table of prime numbers up to 100, but efficiency may become an important consideration, for example, if we want to generate a table of prime numbers up to 1,000,000,000.

Further improvement in efficiency can be realized by noting that a number is prime if it is not evenly divisible by any other prime number. This is because any nonprime integer can be expressed as a multiple of prime factors. (For example, 10 has the prime factors 2 and 5; the nonprime integer 40 has the prime factors 2, 2, 2 and 5.) Using this information, we can develop a more efficient prime number program. The program will test if a given odd integer is prime by determining if it is evenly divisible by any other previously generated prime number. Since previously generated prime numbers will be needed again and again in the program (as opposed to being generated, printed, and then erased from memory), we shall employ an array to store each prime number as it is generated.

Finally, it should be clear that any nonprime integer I must have as one of its factors an integer that is less than or equal to the square root of I. Consequently, as a further optimization of our program, we will test each integer I for even divisibility only by all prime numbers up to the square root of I. Presented below are two FORTRAN programs (PRIME3 and PRIME4) that implement these improvements:

```
      PROGRAM PRIME3
C*************************************************************************
C   Determines and displays prime numbers up to user-entered value N
C*************************************************************************
      IMPLICIT NONE
      INTEGER I, J, N, PINDEX, PRIME(100)
      PRINT*, 'Enter N: '
      READ*, N
      PRINT*, 'Prime numbers up to', N, ':'
      PRINT*, ' '
C
C   Store the first two primes
      PRIME(1) = 2
      PRIME(2) = 3
C
```

```
C  PINDEX is the next free slot in the PRIME array
      PINDEX = 3
C
C   Test odd integer I for divisibility by primes less than or equal to the
C   square root of I.  I is prime if and only if it is not divisible by any
C   such integer.
      DO 1 I = 5, N, 2
          DO 2 J = 1, PINDEX-1
              IF(PRIME(J) .GT. SQRT(REAL(I))) GO TO 4
              IF(MOD(I, PRIME(J)) .EQ. 0) GO TO 1
    2     CONTINUE
    4     PRIME(PINDEX) = I
          PINDEX = PINDEX + 1
    1 CONTINUE
      DO 3 I = 1, PINDEX-1
          PRINT*, PRIME(I)
    3 CONTINUE
      STOP
      END
```

Note that PROGRAM PRIME3 employs two conditional GO TO's (i.e., GO TO 1 and GO TO 4) to exit from the inner loop. Some programmers hold that the use of the GO TO is justified to exit from a nest of loops while executing an inner loop or to start the next iteration of an outer loop from within an inner loop. It may be argued that, in such cases, a program written with the GO TO is easier to comprehend than one without it. Many advocates of structured programming, on the other hand, argue against the use of GO TO's unless such usage is absolutely necessary. PROGRAM PRIME4 is an implementation of exactly the same algorithm used in PROGRAM PRIME3, but it avoids the use of GO TO's.

```
      PROGRAM PRIME4
C*************************************************************************
C   Determines and displays prime numbers up to the user-entered value N
C*************************************************************************
      IMPLICIT NONE
      LOGICAL PRIME
      INTEGER I, J, N, PINDEX, PRIMES(100)
      PRINT*, 'Enter N: '
      READ*, N
      PRINT*, 'Prime numbers up to', N, ':'
      PRINT*, ' '
C
C   Store the first two primes
      PRIMES(1) = 2
      PRIMES(2) = 3
C
C   PINDEX is the next free slot in the PRIMES array
      PINDEX = 3
C
C   Test odd integer I for divisibility by primes less than or equal to the
C   square root of I.  I is prime if and only if it is not divisible by any
C   such integer.
```

```
      DO 1 I = 5, N, 2
          PRIME = .TRUE.
          J = 1
          DO WHILE(PRIMES(J).LE.I/PRIMES(J) .AND. PRIME)
              IF(MOD(I,PRIMES(J)) .EQ. 0) PRIME = .FALSE.
              J = J + 1
          END DO
          IF(PRIME)THEN
              PRIMES(PINDEX) = I
              PINDEX = PINDEX + 1
          ENDIF
    1 CONTINUE
C
C   Print the results
      DO 3 I = 1, PINDEX-1
          PRINT*, PRIMES(I)
    3 CONTINUE
      STOP
      END
```

Exercise:

9. Prime numbers can also be generated by an algorithm called the *Sieve of Erastosthenes*. The algorithm is presented below:

Step 1: Define an array of integers P. Set all elements P_i to 0, i = 2, N.
Step 2: Set i to 2.
Step 3: If i is greater than N, the algorithm terminates.
Step 4: If P_i is zero, than i is prime.
Step 5: For all positive values of j such that ixj is less than or equal to N, set P_{ixj} to 1.
Step 6: Add 1 to i and go to step 3.

Implement this algorithm in FORTRAN. (Do not worry about *why* the algorithm works. You should, however, be able to convert the steps given above to a FORTRAN program. Do not look at the "solution" given below until you finish and test your program.)

Solution:

```
      PROGRAM PRIME5
C*************************************************************************
C   Determines and displays prime numbers up to user-entered value N
C*************************************************************************
      IMPLICIT NONE
      INTEGER I, J, N, P(2:100)
      PRINT*, 'Enter N: '
      READ*, N
      PRINT*, 'Prime numbers up to', N, ':'
      PRINT*, ' '
C
C   Sieve of Erastosthenes Algorithm
      DO 1 I = 2, N
          P(I) = 0
    1 CONTINUE
```

```
      DO 2 I = 2, N
          IF(P(I).EQ.0) PRINT*, I
          DO 3 J = 1, N/I
              P(I*J) = 1
3         CONTINUE
2     CONTINUE
      END
```

3.10 Implied DO Loops

In the example programs given so far, "explicit" DO loops were employed to read in and print out arrays. When using explicit DO loops for printing and reading, a distinct line of output is displayed for each execution of a PRINT statement, and a distinct line of input is required for each execution of a READ. This can sometimes be cumbersome and inconvenient. A usually better approach is the use of **implied DO loops**. For example, the explicit DO loop

```
      DO 1 I = 1, N
          READ*, A(I)
1     CONTINUE
```

can be replaced by the implied DO loop

```
      READ*, (A(I), I = 1, N)
```

When this loop is executed, as many lines as needed will be read until all of the array elements (from A(1) to A(N)) are filled. As many values as desired can be entered on a single line (i.e., without pressing the ENTER key), since a separate line is not required for each data item. The general form of an implied DO loop is as follows:

PRINT*, (*output list, lcv = inv, endv, step*)
READ*, (*input list, lcv = inv, endv, step*)

where *lcv* is the loop-control variable which must be of type integer; the loop parameters *inv, endv,* and *step* denote initial value, end value, and step size (increment), respectively, for the loop-control variable. When *step* is omitted, it is assumed to be +1. For example,

```
      READ*, (A(I), B(I), I = 1, 5, 2)
```

has the same affect as the statement

```
      READ*, A(1), B(1), A(3), B(3), A(5), B(5)
```

In general, the variables in an implied DO loop do not have to be subscripted. It is also possible to PRINT constants using an implied DO loop. It is permissible, for example, to use a statement like

```
PRINT*, ('-', I = 1, 80)
```

instead of the clumsy statement

```
PRINT*, '--------------------------------------------------------
&------------------------'
```

Example 3.19:

Consider the problem of reading and printing the elements of an array. One way to accomplish this is to use explicit DO loops to read and print the array elements one by one. To see how this approach works, type and run the following program:

```
      PROGRAM EXPLDO
C*************************************************************************
C   Program shows how the elements of an array (N data values)
C   are read in and printed out using "explicit" DO loops.
C*************************************************************************
      IMPLICIT NONE
      INTEGER N, I
      REAL A(10)
      PRINT*, 'Enter the number of values: '
      READ*, N
      DO 1 I =1, N
          PRINT*, 'Enter value', I, ':  '
          READ*, A(I)
    1 CONTINUE
      DO 2 I = 1, N
          PRINT*, 'I =', I, '   A(I) =', A(I)
    2 CONTINUE
      END
```

The following are two sample execution outputs of this program:

```
Enter the number of values: 3
Enter value           1:  1.5
Enter value           2:  3.6
Enter value           3:  2.3
I =           1   A(I) =    1.50000
I =           2   A(I) =    3.60000
I =           3   A(I) =    2.30000

Enter the number of values: 4
Enter value           1:  4
Enter value           2:  2    7
Enter value           3:  8
Enter value           4:  7
I =           1   A(I) =    4.00000
I =           2   A(I) =    2.00000
I =           3   A(I) =    8.00000
I =           4   A(I) =    7.00000
```

Notice that, in the second run, two data values (2 and 7) were typed on the same line before the ENTER key was pressed. The second value, however, is ignored by the program because the READ statement

```
READ*, A(I)
```

expects and reads exactly one data value. Furthermore, when the READ statement is executed again in the next iteration of the DO loop, the value in the next line (i.e. 8) is read, and 7 is completely skipped. As a result, after typing the four values 4, 2, 7, and 8, the program still prompts us to enter one more value:

```
Enter the number of values: 4
Enter value           1:  4
Enter value           2:  2    7
Enter value           3:  8
Enter value           4:
```

We next type 7 to complete the array. (This will be acceptable only if the order of the array elements is immaterial.) The use of implied DO loops to read and print arrays is illustrated below. Run this program also with a variety of input data:

```
      PROGRAM IMPLDO
C*************************************************************************
C   Program shows how the elements of an array (N data values)
C   are read in and printed out using implied DO loops.
C*************************************************************************
      IMPLICIT NONE
      INTEGER N, I
      REAL A(10)
      PRINT*, 'Enter the number of values: '
      READ*, N
      PRINT*, 'Enter', N, ' values: '
      READ*, (A(I), I = 1,N)
      PRINT*, 'The values you entered are: '
      PRINT*, (A(I), I = 1,N)
      END
```

Two sample outputs are provided below:

```
Enter the number of values: 3
Enter           3 values: 1  4  2
The values you entered are:
    1.00000        4.00000        2.00000

Enter the number of values: 4
Enter           4 values: -2.4    4
1
3
The values you entered are:
   -2.40000        4.00000        1.00000        3.00000
```

In the second run, the first two values were typed on the first line of input, then ENTER was pressed. The third and the fourth data values were entered on separate lines. This is quite acceptable when an implied DO loop is used.

3.11 Program Parameters

FORTRAN has a special feature, the **PARAMETER statement**, which is helpful in writing programs that are more readable and easier to modify. Consider, for example, a program that contains the following declarations:

```
REAL A(10), B(10), C(10), D(10)
INTEGER NUM(10), INDEX(10)
```

and also program segments such as

```
PRINT*, 'Enter the number of array elements: '
READ*, N
IF(N.GT.10)THEN
    PRINT*, 'Array size N is too large. Execution terminated.'
    STOP
ENDIF
READ*, (A(I), I = 1,N)
```

If a certain application required handling arrays with a dimension higher than 10, it would be necessary to modify this program by changing the number 10 at each and every place it is used as the maximum array size. Alternatively, using the PARAMETER statement, the above program segments can be written as follows:

```
INTEGER NMAX
PARAMETER(NMAX = 10)
REAL A(NMAX), B(NMAX), C(NMAX), D(NMAX)
INTEGER NUM(NMAX), INDEX(NMAX)
 .
 .
PRINT*, 'Enter the number of array elements: '
READ*, N
IF(N.GT.NMAX)THEN
    PRINT*, 'Array size N is too large. Execution terminated.'
    STOP
ENDIF
READ*, (A(I), I = 1,N)
```

With this approach, it would be sufficient to change the value of NMAX in the PARAMETER statement to handle arrays of different sizes. The general format of the PARAMETER statement is as follows:

PARAMETER $(par_1 = const_1, par_2 = const_2, ...)$

This statement associates with each symbolic name par_i with a constant value $const_i$. Each such symbolic name can then be used to represent the associated value elsewhere in the program. Such a symbolic name is referred to as a **parameter** or a **named constant**. The type of each parameter must be declared before it appears in

a PARAMETER statement. PARAMETER statements must be placed before all executable statements in the program. A symbolic name used as a parameter cannot be redefined with an assignment statement, a READ statement, or another PARAMETER statement. (For example, the value of the parameter NMAX discussed above cannot be changed in the program.) A parameter cannot be used before the PARAMETER statement that defines it.

3.12 Multidimensional Arrays

So far we have dealt only with one-dimensional (linear) arrays. These arrays are employed to represent variables with a single subscript. FORTRAN also allows the use of two or three (and higher) dimensional arrays which are useful for representing and manipulating variables with multiple subscripts.

Consider, for example, the problem of adding two MxN matrices A and B. (Remember that a matrix is a rectangular "table" or "collection" of numbers. An MxN matrix contains M rows and N columns.) We denote the number in the i-th row and the j-th column of A by a_{ij}. Let C be defined as the sum of A and B. Then, by the definition of matrix addition, we have $c_{ij} = a_{ij} + b_{ij}$. Assuming that the matrices we will deal with can have at most 10 columns and 10 rows, we would place the following array declarations at the top of our program to allocate space for the matrices A, B, and C in the computer memory:

```
      REAL A(10,10), B(10,10), C(10,10)
```

To calculate the elements of the sum matrix C, from given matrices A and B, we can then include the following segment in our program:

```
      DO 1 I = 1, N
          DO 2 J = 1, M
              C(I,J) = A(I,J) +  B(I,J)
2         CONTINUE
1     CONTINUE
```

An *n-dimensional array declaration*, in its most general form, has the following form:

type name(low_1:$high_1$, low_2:$high_2$,...,low_n:$high_n$)

where *type* may be any of the FORTRAN data types, i.e. INTEGER, REAL, CHARACTER, LOGICAL, etc., and low_k and $high_k$ denote the smallest and largest

subscript values for dimension *k* of the array indicated by *name*. Clearly, low_k has to be less than or equal to $high_k$, and the number of elements of the array is

$$(high_1 - low_1 + 1)x(high_2 - low_2 + 1)x...x(high_n - low_n + 1)$$

A simpler and commonly used form of n-dimensional array declaration is as follows:

type name$(size_1,size_2,...,size_n)$

where $size_k$ has to be a positive integer. When an array is declared in this way, the smallest subscript for each dimension is automatically assumed to be +1. The largest allowable subscript for dimension *k* is then equal to $size_k$.

Example 3.20:

Let A and B be real matrices with dimensions $N_A x M_A$ and $N_B x M_B$, respectively. The product AB is defined if $M_A = N_B$ and is an $N_A x M_B$ matrix, say C, which is calculated as follows:

$$c_{ij} = \Sigma_k (a_{ik} b_{kj})$$

where the summation is from k=1 to $k=M_A$ $(=N_B)$. Similarly, the product BA, call it D, is well-defined when $M_B = N_A$ and is an $N_B x M_A$ matrix:

$$d_{ij} = \Sigma_k (b_{ik} a_{kj})$$

where the sum is taken from k=1 to $k=M_B$ $(=N_A)$. Given two user-entered matrices A and B, the following program computes and prints the products AB and BA, if they are defined.

```
      PROGRAM MATMUL
C*************************************************************************
C   Program to carry out matrix multiplication:
C   Matrices A and B are read in interactively, product matrices
C   C = A B and D = B A, if they exist, are computed and printed out.
C*************************************************************************
      IMPLICIT NONE
      INTEGER NMAX
      PARAMETER(NMAX = 10)
      REAL A(NMAX,NMAX), B(NMAX,NMAX), C(NMAX,NMAX), D(NMAX,NMAX)
      INTEGER I, J, K, NA, NB, MA, MB
      LOGICAL CEXIST, DEXIST
      PRINT*, 'Program will calculate C = AB and D = BA.'
      PRINT*, 'Enter the dimensions of A: '
      READ*, NA, MA
      IF(NA.GT.NMAX .OR. MA.GT.NMAX)THEN
          PRINT*, 'A is too large. Execution terminated.'
          STOP
      ENDIF
      DO 1 I = 1, NA
          PRINT*, 'Enter row number', I, ':  '
          READ*, (A(I, J), J = 1, MA)
    1 CONTINUE
```

```
      PRINT*, 'Enter the dimensions of B: '
      READ*, NB, MB
      IF(NB.GT.NMAX .OR. MB.GT.NMAX)THEN
          PRINT*, 'B is too large. Execution terminated.'
          STOP
      ENDIF
      DO 2 I = 1, NB
          PRINT*, 'Enter row number', I, ':  '
          READ*, (B(I, J), J = 1, MB)
    2 CONTINUE
C
C   Calculate the elements of the product matrix C
      IF(NB.EQ.MA)THEN
          CEXIST = .TRUE.
          DO 3 I = 1, NA
              DO 4 J = 1, MB
                  C(I,J) = 0.0
                  DO 5 K =1, MA
                      C(I,J) = C(I,J) + A(I,K)*B(K,J)
    5             CONTINUE
    4         CONTINUE
    3     CONTINUE
      ELSE
          PRINT*, 'The product AB is not defined. '
          CEXIST = .FALSE.
      ENDIF
C   Calculate the elements of the product matrix D
      IF(NA.EQ.MB)THEN
          DEXIST = .TRUE.
          DO 6 I = 1, NB
              DO 7 J = 1, MA
                  D(I,J) = 0.0
                  DO 8 K =1, MB
                      D(I,J) = D(I,J) + B(I,K)*A(K,J)
    8             CONTINUE
    7         CONTINUE
    6     CONTINUE
      ELSE
          PRINT*, 'The product BA is not defined. '
          DEXIST = .FALSE.
      ENDIF
C   Print results
      IF(CEXIST)THEN
          PRINT*, ' '
          PRINT*, 'Product AB:'
          DO 9 I = 1, NA
              PRINT*, (C(I, J), J = 1, MB)
    9     CONTINUE
      ENDIF
      IF(DEXIST)THEN
          PRINT*, ' '
          PRINT*, 'Product BA:'
          DO 10 I = 1, NB
              PRINT*, (D(I, J), J = 1, MA)
   10     CONTINUE
      ENDIF
      END
```

Here is a sample run of this program:

```
Program will calculate C = AB and D = BA.
Enter the dimensions of A: 2  2
Enter row number           1:  1  0
Enter row number           2:  2  3
Enter the dimensions of B: 2  2
Enter row number           1:  1  5
Enter row number           2:  0  2

Product AB:
    1.00000         5.00000
    2.00000         16.0000

Product BA:
    11.0000         15.0000
    4.00000         6.00000
```

3.13 The DATA Statement

Most programs require certain input data (which are entered either using the keyboard or read in from a data file), carry out calculations (during which intermediate data may be generated), and produce output (which are normally displayed on the computer screen or written into an output file). Input data may change from run to run, and accordingly the outputs obtained during different runs may also be different. Occasionally we need to define certain data values, which are not expected to change from one execution of the program to the next, at the time we write the program. It is unnecessary to use READ statements to load such data into memory. The **DATA statement** is usually employed to define these data values. The general form is

DATA *List of variables* */List of constants/*

The DATA statement is a non-executable statement and should be placed after the type declaration statements. It is common practice to place the DATA statements before all executable statements. The following are some valid examples of the DATA statement:

```
DATA A /1.0/
DATA X, Y /3.5, 200./, FLAG /.TRUE./
DATA NAME(1), NAME(2) /'Ahmet', 'Nazan'/
```

The use of the above DATA statements has the same effect as the execution of the following assignment statements:

```
A = 1.0
X = 3.5
Y = 200.
```

```
FLAG = .TRUE.
NAME(1) = 'Ahmet'
NAME(2) = 'Nazan'
```

The difference is, the DATA statement assigns values to variables during compilation, i.e. before execution of the program. One consequence of this is that, as a result of using DATA statements (instead of the executable assignment statements), the execution time of the program is reduced.

Recall that the PARAMETER statement, which is also a non-executable statement, can be used to associate constant values with symbolic names. For example, the statement

```
PARAMETER(PI = 3.14159)
```

associates the symbolic name PI with the value 3.14159. Here PI is *not* a variable and cannot be assigned a value in the program. The DATA statement, on the other hand, is employed to assign values to variables which, if desired, can be changed later in the program. For example, the DATA statement is frequently employed to initialize arrays. Let A be an array declared as

```
REAL A(10)
```

then the (initial) values 11.0, -5.0, 3.5, 0.0, 1.0 can be assigned to A(1), A(2), A(3), A(4), and A(5), respectively, as follows:

```
DATA (A(I), I=1,5) /11.0, -5.0, 3.5, 0.0, 1.0/
```

If all of the elements of A are to be initialized to zero, then any one of the following statements can be used:

```
DATA (A(I),I=1,10)/0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0/
DATA (A(I), I=1,10) /10*0.0/
DATA A /0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0/
DATA A /10*0.0/
```

It should be noted that there are as many constants as there are memory cells being initialized. To repeat a constant n times, the notation n*c is used (where c represents the constant).

Example 3.21:

Consider a telephone survey carried out to discover how viewers feel about a particular television show. Each respondent is asked to rate the show on a scale from 1 to 5. The following FORTRAN program can be used to analyze the results of the survey. The program determines how many people rated the show a 1, how many a 2, and so on up to 5. Note that the maximum rating point is specified using the parameter MAXRAT which can be changed if desired. If the rating scale is 1 to 10, for example, MAXRAT will be 10.

Type and run the program. While the program could be used to analyze the results of, say, 5000 responses, use small test cases (e.g. 10 to 20 responses) to evaluate the program.

```
      PROGRAM RATING
C*************************************************************************
C  Program to analyze ratings of a TV show
C  N         : number of responses from viewers
C  MAXRAT    : maximum rating point
C  COUNTR(I) : # of people rating the show an I (I=1,MAXRAT)
C*************************************************************************
      IMPLICIT NONE
      INTEGER MAXRAT
      PARAMETER(MAXRAT=5)
      INTEGER COUNTR(MAXRAT), I, N, RESPON
      DATA COUNTR /MAXRAT*0/
      PRINT*, 'Enter number of responses: '
      READ*, N
      PRINT*, 'Enter your responses: '
      I = 1
      DO WHILE(I.LE.N)
          READ*, RESPON
          IF(RESPON.LT.1 .OR. RESPON.GT.MAXRAT)THEN
              PRINT*, 'Bad response:', RESPON
          ELSE
              COUNTR(RESPON) = COUNTR(RESPON) + 1
              I = I + 1
          ENDIF
      END DO
      PRINT*, 'Rating  Number of Responses'
      PRINT*, '---------------------------'
      DO 1 I = 1, MAXRAT
          PRINT '(1X,I3,7X,I4)', I, COUNTR(I)
    1 CONTINUE
      END
```

Example 3.22:

Consider the factorial function *n! = (n)(n-1)(n-2)...(1)*. The following straightforward function can be employed to calculate the factorial of a given integer *n*. Note that the function is typed as REAL to avoid integer overflow when *n* is large.

```
      REAL FUNCTION FACT(N)
      IMPLICIT NONE
      INTEGER N, I
      FACT = 1.0
      DO 10 I = 2, N
          FACT = FACT*I
   10 CONTINUE
      RETURN
      END
```

Suppose, however, that this function will be called a very large number of times in a program, and the factorials of some or all of the smaller integers (say, those between 0 and 10) will be needed repeatedly. With the use of the above function, the mentioned factorials would have to be re-calculated every time they are needed. A more efficient approach (one that may decrease execution time) is to set up a table that holds the factorials that the calling program is likely to need frequently. Here is how this can be done using the DATA statement[5]:

```
      REAL FUNCTION FACT(N)
      IMPLICIT NONE
      INTEGER N, I
      REAL TABLE(0:10)
      DATA TABLE /1,1,2,6,24,120,
     1            720,5040,40320,
     2            362880,3628800/
      IF(N.LE.10)THEN
          FACT=TABLE(N)
          RETURN
      ENDIF
      FACT=TABLE(10)
      DO 10 I = 11, N
          FACT = FACT*I
   10 CONTINUE
      RETURN
      END
```

For $n \leq 10$, $n!$ is simply looked up in the table, saving $(n-1)$ multiplications. If $n>10$, $n!$ is calculated as $10!(11)(12)...(n-1)(n)$, saving 9 multiplications. An alternative approach (using the ENTRY statement) is given in Example 4.15.

Note also how the RETURN statement is utilized to end the execution of the subprogram before reaching the END statement. (Remember that the RETURN statement just before the END statement is optional and could be omitted. The first RETURN statement, however, is essential for the function to work correctly.)

Exercises:

10. Example 3.21 allows a variable number of responses (N) which is entered by the user. Modify the program so that the user does not have to count the number of responses in the list. Set up the program such that the value 999 can be typed in by the user to indicate that the last response has been entered.

11. An algorithm for calculating square roots is as follows. If x_{old} is a guess for the value of the square root of a number C, then an improved value is x_{new}, where

$$x_{new} = (x_{old} + C/x_{old})/2$$

The idea is to read a number C whose square root is to be computed, guess the square root x_{old} (you may use (C+1)/2 as the initial guess), and improve the guess with the above equation. If the improved guess x_{new} is not accurate enough (i.e. if $|(x_{new})^2-C|$ is not small, e.g. less than 10^{-6}), the value is again improved by a further application of the algorithm. This process is continued until the computed value for x_{new} is sufficiently accurate or until the program exceeds its maximum cycle limit NMAX. The maximum number of iterations NMAX is supplied by the programmer. If the condition $|(x_{new})^2-C| \leq 10^{-6}$ is satisfied before the maximum number of iterations is reached, the root search is successful. Write a program that will carry out these calculations.

---

[5] Kruger, *Efficient FORTRAN Programming*, p.24.

12. An algorithm to compute the inverse of a number C without using any division is[6]

$$x_{new} = x_{old} (2 - Cx_{old})$$

provided that the initial guess for the inverse (x_{old}) is chosen such that ($2 - Cx_{old}$) is positive. Write a program that will

(i) Prompt and read a positive number C.
(ii) Specify an initial guess x for the inverse of C.
(iii) Check that (2-Cx) > 0. If not, reduce x and try again.
(iv) Prompt and read the maximum number of iterations `NMAX`.
(v) Carry out successive iterations to improve the guess by the given algorithm. The test for success is $|1-Cx| < 10^{-6}$.
(vi) If successful, print the number C and its estimated inverse; if not successful, print a diagnostic (a message explaining what happened).

[6] This exercise has been adapted from *FORTRAN 77 and Numerical Methods for Engineers* by Borse.