Modelli Numerici in Fisica

Appunti di Fortran

da

www.strath.ac.uk/CC/Courses/fortran.html

Constants and Variables

Data are constants provided to a program for use in computation (processing).

Results are constants produced as a result of computation.

We have seen that all information is represented in the computer in binary form. The **type** of information determines the way in which it is represented and the operations which may be performed on it.

The CHARACTER type

A constant of type **CHARACTER,** (often called a **string)** is a sequence of characters which may be upper case alphabetic, numeric, blanks, and the following:

+ - * / = ( ) , . ' $:

When included in a FORTRAN statement, a string must be delimited by single quotes ('). A single quote may be included in a string by writing two consecutively. Only one is retained.

Example: 'WE"RE A" JOCK TAMSON"S BAIRNS.'

The INTEGER type

Constants of type **INTEGER** are integer numbers. An INTEGER constant is written as a sequence of decimal digits, optionally preceded by a sign (unary + or -).

Examples: 123 +1 0 4356 -4

INTEGER constants are represented in **exact** form. Their magnitude has a limit which depends on the word length of the computer.

The REAL type

Constants of type **REAL** are numbers which may include a fractional part. A REAL constant is written in one of the following forms:

1. An integer part written as an INTEGER constant defined as above, followed by a decimal point, followed by a fractional part written as a sequence of decimal digits. Either the integer or the fractional part, but not both, may be omitted.
2. An INTEGER constant or a REAL constant defined as in (i), followed by a decimal **exponent** written as the letter 'E' followed by an INTEGER constant. The constant is a power of 10 by which the preceding part is multiplied.

Examples:

+123.4 -123.4 .6E-3 (0.6x10-3) 4.6E3 (4.6x103) 7E-3 2.

REAL constants are represented in approximate form. Their magnitude has a limit which depends on the word length of the computer.

Variables

A **variable** is a unique name which a FORTRAN program applies to a word of memory and uses to refer to it. A variable consists of one to six upper case alphabetic characters and decimal digits, beginning with an alphabetic character.

Examples:

```
VOL     TEMP    A2      COLUMN  IBM370
```

Note:

Spaces are ignored by FORTRAN, e.g. '`COL UMN`' is equivalent to '`COLUMN`'

1. Clarity can be improved by choosing variables which suggest their usage, e.g.

```
DEGC MEAN STDDEV
```

The **value** of a variable is the constant stored in the word to which it refers. Each variable has a **type,** which stipulates the type of value it may have. The type of a variable may be specified explicitly, or assigned implicitly (by default).

Explicit typing

The type of a variable may be assigned explicitly by a **type specification** statement. This has the form:

type variable_list

where *type* is the name of a type

and *variable_list* is a single variable or a list of variables, separated by commas.

The statement assigns the given type to all the variables in the list.

Examples:

```
INTEGER WIDTH

REAL NUM, K
```

Type specification statements are not compiled into executable machine code instructions. Instead the compiler records the names and types of the variables and reserves storage for them. Such **non-**

executable statements must be placed at the beginning of a program, before the first executable statement.

Implicit (or default) typing

If a variable is used without being included in a type specification, its type is assigned implicitly (by default) according to the following rule:

If the variable begins with a character from `I` to `N`, its type is INTEGER. Otherwise, it is REAL.

Thus `TEMP` is a REAL variable, while `ITEMP` is an INTEGER.

Note: Because a variable can be used without first being declared in a type specification, a misspelled variable is not in general detected as an error by the compiler. The program may compile and run, but produce incorrect results. Care should therefore be taken to get variable names right, and if unexpected results are obtained, variable names are one of the first things to check.

Assigning a value

Before a variable can be used in computation, it must be assigned an initial value. This may be done by reading a value from input or by using an assignment statement.

The READ statement

The **READ** statement is used to assign values to variables by reading data from input. The simplest form of the READ statement is:

`READ` *, *variable_list*

where *variable_list* is a single variable or a list of variables separated by commas. (The asterisk will be explained later).

This statement reads constants from the terminal, separated by spaces, commas, or new lines, and assigns them in sequence to the variables in the list. Execution of the program pauses until the right number of constants has been entered.

Example:

```
READ *, VAR1, VAR2, VAR3
```

waits for three constants to be entered and assigns them in sequence to the variables `VAR1`, `VAR2` and `VAR3`.

The assignment statement

The simplest form of assignment statement is:

variable = *constant*

This means that the constant is assigned as a value to the variable on the left-hand-side. Note that the '=' sign has a different meaning than in algebra. It does not indicate equality, but is an **assignment operator**.

Examples:

```
TEMP = 74.5

ITEMP = 100
```

Type rules

Whichever method is used to assign a value to a variable, the type of the value must be consistent with that of the variable. The rules are:

1. A CHARACTER value cannot be assigned to a numeric variable or vice versa.
2. An INTEGER value can be assigned to a REAL variable. The value assigned is the REAL equivalent of the integer.

Example: X = 5 is equivalent to X = 5.0

1. A REAL value can be assigned to an INTEGER variable. The value assigned is **truncated** by discarding the fractional part.:

Examples:

```
                Value
                Assigned


N = 0.9999      0

M = -1.9999     -1
```

Arrays

All our programs so far have required the storage of only a few values, and could therefore be written using only a few variables. For example, the average mark program of Figure 8 on page 18 required only variables for a mark, the total mark, the count of the marks and the average. When large numbers of values have to be stored, it becomes impracticaI or impossible to use different variables for them all. If the average mark program were rewritten to compute average marks for five subjects, we should require five variables, say MARK1 ... MARK5 for the marks, five variables for the totals, and five for the averages. This could be done, but the program would be rather repetitive. The situation is even worse if, after computing the averages, the program is required to print a list showing, for each student and subject, the student's mark and the difference between the mark and the average. This could conceivably be done if the number of students were given in advance, but the program would be extremely cumbersome. If, as in the example, the number of students is not given but determined by counting, the task is impossible, as there is no way of knowing how many variables will be required.

We need to store all the marks in order in a list or other structure to which we can apply a name, and refer to individual marks by a combination of the name and a number or numbers indicating the position of a mark in the list or structure.

In mathematics, an ordered list of items is called a **vector** of **dimension** . If the vector is denoted by , the items, usually called the **components** or **elements** of the vector, are denoted by .

FORTRAN uses a structure similar to a vector called an **array**. An array **A** of dimension **N** is an ordered list of `N` variables of a given type, called the **elements** of the array. In FORTRAN, the subscript notation used for the components of a vector is not available. Instead the elements are denoted by the name of the array followed by an integer expression in parentheses. Thus, the elements of A are denoted by `A(1), A(2),... A(N).` The parenthesised expressions are called **array subscripts** even though not written as such.

A subscript can be any arithmetic expression which evaluates to an integer. Thus, if `A, B`, and `C` are arrays, the following are valid ways of writing an array element:

```
A(10)
B(I+4)
C(3*I+K)
```

Array declarations

Since an array is a list of variables, it obviously requires several words or other units of storage. Each array must therefore be declared in a statement which tells the compiler how many units to reserve for it. This can be done by including the array name in a type specification followed by its dimension in parentheses. For example:

```
INTEGER AGE(100),NUM(25),DEG
```

This reserves 100 words of storage for array AGE, 25 for array NUM, and one word for the variable DEG. All three items are of type INTEGER.

Space can also be reserved for arrays by the **DIMENSION** statement, which reserves storage using a similar syntax, but includes no information about type. Thus, if this method is used, the type is either determined by the initial letter of the array or assigned by a separate type specification. Therefore, the equivalent to the above using a DIMENSION statement is:

```
INTEGER AGE,DEG
DIMENSION AGE(100),NUM(25)
```

(NUM is typed as INTEGER by default).

DIMENSION statements, like type specifications, are non-executable and must be placed before the first executable statement.

Since a type specification can stipulate both type and dimension, there is little point in using DIMENSION statements.

When this form of declaration is used in a type or DIMENSION statement the upper and lower bounds for the subscript are 1 and the dimension respectively. Thus, AGE in the above example may have any subscript from 1 to 100. Arrays can also be declared to have subscripts with a lower bound other than 1 by using a second form of declaration in which the lower and upper bounds are given, separated by a colon. For example:

```
REAL C(0:20)
INTEGER ERROR(-10:10)
```

reserves 21 words of storage for each of the arrays C and ERROR and stipulates that the subscripts of C range from 0 to 20 inclusive, while those of ERROR range from -10 to 10.

Although the declaration stipulates bounds for the subscript, not all compilers check that a subscript actually lies within the bounds. For example, if NUM is declared as above to have a subscript from 1 to 25, a reference to NUM(30) may not cause an error. The compiler may simply use the 30th word of storage starting from the address of NUM(1) even though this is outside the bounds of the array. This can cause unpredictable results. Care should therefore be taken to make sure that your subscripts are within their bounds.

Use of arrays and array elements

Array elements can be used in the same way as variables, their advantage being that different elements of an array can be referenced by using a variable as a subscript and altering its value, for example by making it the control variable of a DO loop. This is illustrated in the following sections.

The array name without a subscript refers to the entire array and can be used only in a number of specific ways.

Initialising an array

Values can be assigned to the elements of an array by assignment statements, e.g.

```
        NUM(1) = 0

NUM(2) = 5
```

If all the elements are to have equal values, or if their values form a regular sequence, a DO loop can be used. Thus, if NUM and DIST are arrays of dimension 5:

```
        DO 10, I = 1,5
        NUM(I) = 0
        10      CONTINUE
```

initialises all the elements of `NUM` to 0, while:

```
        DO 10, I = 1,5
        DIST(I) = 1.5*I
        10      CONTINUE
```

assigns the values 1.5, 3.0, 4.5, 6.0 and 7.5 to `DIST(1)`,`DIST(2)`,`DIST(3)`, `DIST(4)` and `DIST(5)` respectively.

The DATA statement

The DATA statement is a non-executable statement used to initialise variables. It is particularly useful for initialising arrays. It has the form:

`DATA` *variable_list*/*constant_list*/ [,*variable_list*/*constant_list*/] ...

(The square brackets and ellipsis have their usual meaning.)

Each *variable_list* is a list of variables, and each *constant_list* a list of constants, separated by commas in each case. Each *constant_list* must contain the same number of items as the preceding *variable_list* and corresponding items in sequence in the two lists must be of the same type.

The DATA statement assigns to each variable in each *variable_list* a value equal to the corresponding constant in the corresponding *constant_list*. For example:

```
        DATA A,B,N/1.0,2.0,17/
```

assigns the values 1.0 and 2.0 respectively to the REAL variables `A` and `B`, and 17 to the INTEGER variable `N`.

A constant may be repeated by preceding it by the number of repetitions required (an integer) and an asterisk. Thus:

```
DATA N1,N2,N3,N4/4*0/
```

assigns a value of zero to each of the variables `N1,N2,N3` and `N4`.

Items in a *variable_list* may be array elements. Thus, if `A` is an array of dimension 20, the DATA statement:

```
DATA A(1),A(2),A(3),A(4)/4*0.0/,A(20)/-1.0/
```

assigns a value of zero to the first four elements, -1.0 to the last element, and leaves the remaining elements undefined.

The implied DO list

When a large number of array elements have to be initialised, we can avoid writing them all individually by using an ***implied*** **DO list**.

An implied DO list is used in a DATA statement or an input/output statement to generate a list of array elements. The simplest form of implied DO list is:

(*dlist, int=c1,c2*[,*c3*])

where *dlist* is a list of array elements separated by commas. The expresssion: *int=c1,c2*[,*c3*] has a similar effect to the expression: *var=e1,e2,*[,*e3*] in a DO loop, but *int* must be a variable of type INTEGER, and *c1,c2* and *c3* must be constants or expressions with constant operands. The **implied DO variable** *int* is defined only in the implied DO list, and is distinct from any variable of the same name used elsewhere.

The implied DO list expands *dlist* by repeating the list for each value of *int* generated by the loop, evaluating the array subscripts each time. Thus:

```
DATA (A(I),I=1,4)/4*0.0/,A(20)/-1.0/
```

has the same effect as the previous example.

A more complex use of an implied DO list is shown by the example:

```
DATA (A(I),A(I+1),I=1,19,3)/14*0.0/,(A(I),I=3,18,3)/6*1.0/
```

which assigns a value of zero to `A(1),A(2), A(4),A(5), ... A(19),A(20)` and a value of 1.0 to every third element `A(3),A(6), ... A(18)`.

Finally, an entire array can be initialised by including its name, without a subscript, in *variable_list* in a DATA statement. This is equivalent to a list of all its elements in sequence. Thus, if `A` has dimension 20, all the elements of `A` are initialised to zero by:

```
DATA A/20*0.0/
```

DATA statements can be placed anywhere in a program after any specifications. In the interests of clarity, it is probably best to put them immediately before the first executable statement. Wherever they may be, they cause initialisation *when the program is loaded* (before execution begins). Therefore they can only be used to *initialise* variables and not to re-assign values to them throughout execution of the program. For this purpose, assignment statements or READ statements must be used.

Input and output of arrays

Array elements and array names can be used in input/output statements in much the same way as in DATA statements. Thus, input and output lists can include:

- array elements.
- array names (equivalent to all the elements in sequence).
- implied DO lists.

Implied DO lists in input/output statements differ in two respects from those in DATA statements:

1. In output statements, *dlist* can include *any* output list item. For example:

```
PRINT *, (A(I),'ABC', K, I=1,4)
```

will print the values of `A(1)`...`A(4)` followed in each case by '`ABC`' and the value of `K`.

1. The loop parameters need not be constants or constant expressions, but can include variables (INTEGER or REAL) provided that these have been assigned values, e.g.

```
N = 5
.
PRINT *,(A(I),I=1,N)
```

In an input statement, the loop parameters can depend on values read before by the same statement, e.g.

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

If variables are used in this way, care should be taken to ensure that they lie within the subscript bounds of the array, as in the following example:

```
REAL A(20)
.
READ *, N
IF (N.GE.1 .AND. N.LE.20) THEN
READ *, (A(I),I=1,N)
ELSE
PRINT *, N, 'EXCEEDS SUBSCRIPT BOUNDS.'
END IF
```

We can now return to the exam marks problem mentioned at the beginning of the chapter.

Example 1:

Write a program to read the marks of a class of students in five papers, and print, for each paper, the number of students sitting it and the average mark. The marks are to be read as a list of five marks in the same order for each student, with a negative mark if the student did not sit a paper. The end of the data is indicated by a dummy mark of 999.

The outline of the program is:

1. Initialise the total mark for each of the five papers and a count of the number of students sitting it.
2. Read five marks for the first student.
3. While the first mark is not 999, repeat:
    1. For each of the five marks repeat:
        1. If the mark is not negative then:
            1. Increment the count of students sitting that paper.
            2. Add the mark to the total for that paper.
    2. Read five marks for the next student.
4. Repeat for each of five papers:
    1. If the count of students sitting the paper exceeds zero then:
        1. Compute the average mark for the paper.
        2. Print the number of the paper, the number of students sitting it, and the average mark.

Otherwise

1. Print a message: 'No students sat paper number' *paper_number*

We shall use arrays `MARK`, `COUNT` and `TOTAL` to store the five marks for a student, a count of students sitting each paper and the total mark for each paper respectively. The program follows.

```
        PROGRAM EXAM
        INTEGER MARK(5),TOTAL(5),COUNT(5)
        DATA COUNT/5*0/,TOTAL/5*0/
        READ *,(MARK(I),I=1,5)
        10      IF (MARK(1).NE.999) THEN
                DO 20, I=1,5
                IF (MARK(I).GE.0) THEN
        COUNT(I) = COUNT(I)+1
                TOTAL(I) = TOTAL(I)+MARK(I)
                END IF
        20      CONTINUE
        READ *,(MARK(I),I=1,5)
        GOTO 10
        END IF
        DO 30, I=1,5
        IF (COUNT(I).GT.0) THEN
        AVMARK = 1.0*TOTAL(I)/COUNT(I)
C MULTIPLY BY 1.0 TO CONVERT TO REAL AND AVOID TRUNCATION
        PRINT *,COUNT(I),' STUDENTS SAT PAPER NUMBER',I
        PRINT *,'THE AVERAGE MARK WAS', AVMARK
        ELSE
        PRINT *,'NO STUDENTS SAT PAPER NUMBER',I
        END IF
        30      CONTINUE
```

```
      END
```

Figure 12: Exam marks program

One problem with this program is that if the last line of input consists of the single terminating value of 999, the statement: `READ *,(MARK(I),I=1,5)` will wait for another four values to be entered. This can be avoided by following 999 by a '/' character, which is a terminator causing the READ statement to ignore the rest of the input list.

Multi-dimensional arrays

Suppose now that the exam marks program is to be altered to print a list of all the marks in each paper, with the differences between each mark and the average for the paper. This requires that all the marks should be stored. This could be done by making the dimension of MARK large enough to contain all the marks, and reserving the first five elements for the first student's marks, the next five for the second student's marks and so on. This would be rather awkward.

The problem could be dealt with more easily if we could add a second subscript to the `MARK` array to represent the number of each student in sequence. Our array could then be declared either by:

```
      INTEGER MARK(5,100)
```

or by:

```
      INTEGER MARK(100,5)
```

and would reserve enough space to store the marks of up to 100 students in 5 subjects.

In fact, FORTRAN arrays can have up to *seven* dimensions, so the above declarations are valid. The subscript bounds are specified in the same way as for one-dimensional arrays. For example:

```
      REAL THREED(5,0:5,-10:10)
```

declares a three-dimensional array of type `REAL`, with subscript bounds of 1...5, 0...5 and -10...10 in that order.

An array element must always be written with the number of subscripts indicated by the declaration.

When multi-dimensional array elements are used in an implied DO list, multiple subscripts can be dealt with by including *nested implied DO lists* in *dlist*, for example:

```
      READ *, (A(J),(MARK(I,J),I=1,5),J=1,100)
```

Here, *dlist* contains two items, `A(J)` and the implied DO list `(MARK(I,J),I=,5)` .This inner implied DO list is expanded once for each value of `J` in the outer implied DO list. Thus the above READ statement reads values into the elements of `A` and `MARK` in the order:

```
A(1),   MARK(1,1),      MARK(2,1),...   MARK(5,1)
A(2),   MARK(1,2),      MARK(2,1),...   MARK(5,2)
                .
A(100),MARK(1,100),MARK(2,100),...MARK(5,100)
```

The unsubscripted name of a multi-dimensional array can be used, like that of a one-dimensional array, in input/output and DATA statements to refer to all its elements, but it is essential to know their order. The elements are referenced *in the order of their positions in the computer's memory*. For a one-dimensional array, the elements occur, as we might expect, in increasing order of their subscripts, but for multi-dimensional arrays, the ordering is less obvious. The rule is that the elements are ordered with the first subscript increasing most rapidly, then the next and so on, the last subscript increasing most slowly. Thus if `MARK` is declared as:

```
INTEGER MARK(5,100)
```

its elements are ordered in memory as shown above, and the statement:

```
READ *,MARK
```

is equivalent to:

```
READ *, ((MARK(I,J),I=1,5),J=1,100)
```

Of course, the order could be altered by swapping the control variables in the inner and outer implied DO loops thus:

```
READ *, ((MARK(I,J),J=1,100),I=1,5)
```

We can use a two-dimensional array to solve the problem posed at the beginning of this section.

Example 2:

Write a program to read the marks of up to 100 students in five papers, and print, for each paper, the number of students sitting it, the average mark, and a list of the marks and their differences from the average. The marks are to be read as a list of five marks in the same order for each student, with a negative mark if the student did not sit a paper. The end of the data is indicated by a dummy mark of 999.

The outline is:

1. Initialise the total mark for each of the five papers, a count of the number of students sitting it and a count of all the students.
2. For up to 100 students, repeat:
    1. Read and store five marks
    2. If the first mark is 999, then continue from step 4.

Otherwise:

1. Increment the count of all students.

2. For each of the five marks repeat:
    1. If the mark is not negative then:
        1. Increment the count of students sitting that paper.
        2. Add the mark to the total for that paper.

1. Read a mark. If it is not 999 then:
    1. Print a message: 'Marks entered for more than 100 students.'
    2. STOP
2. Repeat for each of five papers:
    1. If the count of students sitting the paper exceeds zero then:
        1. Compute the average mark for the paper.
        2. Print the number of the paper, the number of students sitting it, and the average mark.
        3. Print a list of all the marks in that paper and their differences from the average for the paper.

Otherwise

1. Print a message: 'No students sat paper number' *paper_number*

Step 4.1.3 can be further outlined as:

1. For each student, repeat:

If his/her mark in the paper is not negative, then:

1. Print the mark.
2. Compute and print the difference between the mark and the average for the paper.

Since the marks are read five subjects at a time for each student, it is convenient to store them in an array `MARK(5,100)`. The program follows:

```
PROGRAM EXAM2
INTEGER MARK(5,100),TOTAL(5),COUNT(5),ALL
DATA COUNT/5*0/,TOTAL/5*0/,ALL/0/
DO 20, J=1,100
READ *,(MARK(I,J),I=1,5)
IF (MARK(1,J).EQ.999) GOTO 30
ALL = ALL+1
        DO 10, I=1,5
        IF (MARK(I,J).GE.0) THEN
COUNT(I) = COUNT(I)+1
        TOTAL(I) = TOTAL(I)+MARK(I,J)
        END IF
10      CONTINUE
20      CONTINUE
        READ *,LAST
        IF (LAST.NE.999) THEN
                PRINT *,'MARKS ENTERED FOR MORE THAN 100 STUDENTS.'
                STOP
        END IF
30      DO 50, I=1,5
IF (COUNT(I).GT.0) THEN
AVMARK = 1.0*TOTAL(I)/COUNT(I)
```

```
C MULTIPLY BY 1.0 TO CONVERT TO REAL AND AVOID TRUNCATION
        PRINT *,COUNT(I),' STUDENTS SAT PAPER NUMBER',I
        PRINT *,'THE AVERAGE MARK WAS', AVMARK
        PRINT *,'MARKS AND THEIR DIFFERENCES FROM THE AVERAGE:'
        DO 40, J=1,ALL
                IF (MARK(I,J).GE.0)PRINT *,MARK(I,J),MARK(I,J)-AVMARK
        40      CONTINUE
        ELSE
        PRINT *,'NO STUDENTS SAT PAPER NUMBER',I
        END IF
        50      CONTINUE
        END
```

Figure 13: Exam marks program (version 2)

Functions and subroutines

Very often, a program has to perform a computation several times using different values, producing a single value each time. An example is the conversion of an angle in degrees to an equivalent in radians in Example 1 of the previous chapter.

In FORTRAN, such a computation can be defined as a **function** and referred to by a name followed by a list of the values (called **arguments)** which it uses, in parentheses, i.e.

name([*argument_list*])

where *argument_list* is an optional list of arguments separated by commas. Note that the parentheses must be included even if *argument_list* is omitted, i.e.

name()

Such a function reference can be used in the same way as a variable or array element, except that it cannot be the object of an assignment. Like a variable or array element, a function reference is evaluated and the value obtained is substituted for it in the expression in which it appears. The **type** of a function is the type of the value so obtained.

Thus, in the above example, a REAL function DGTORD might be defined to convert an angle in degrees to an equivalent in radians. The function would have a single argument, of type INTEGER, representing the value of the angle in degrees, and would be evaluated to obtain the equivalent in radians. The function might be used in an assignment statement like:

```
RADIAN = DGTORD(DEGREE)
```

The definition of a function must include a definition of its type and the number and types of its arguments. In a function reference the number and type of the arguments must be as defined. Thus, for example:

```
RADIAN = DGTORD(DEGREE,X)
```

would be an error.

As the above example illustrates, a function reference has an identical form to an array element, and may be used in a similar context. FORTRAN distinguishes between the two by checking whether the name has been declared as an array, and assuming that it is a function if it has not. Thus, for example, if DGTORD were declared as:

```
REAL DGTORD(100)
```

then `DGTORD(DEGREE)` would be interpreted as an array element and not a function reference.

Intrinsic functions

FORTRAN provides a wide range of **intrinsic** functions, which are defined as part of the language. Many of them have an argument, or list of arguments, which may be of different types in different references. Most, though not all, of these return a value of the same type as that of their arguments in any reference. For example, the function **ABS** returns the absolute value of its argument, which may be REAL or INTEGER. Thus

```
ABS(X)
```

returns the absolute value of the REAL variable `X` as a REAL value, while

```
ABS(N)
```

returns the absolute value of the INTEGER variable `N` as an INTEGER value.

A function of this kind is called a **generic** function. Its name really refers to a group of functions, the appropriate one being selected in each reference according to the type of the arguments.

Figure 18 is a list of some of the more frequently used intrinsic functions. `I` and `R` indicate INTEGER and REAL arguments respectively. Where an argument represents an angle, it must be in radians.

Name	**Type**	**Definition**
ABS(IR)	Generic	Absolute value: IR
ACOS(R)	REAL	arccos(R)
AINT(R)	REAL	Truncation: REAL(INT(R))
ANINT(R)	REAL	Nearest whole number: REAL(INT(R+0.5)) if R0 REAL(INT(R0.5)) if R0
ASIN(R)	REAL	arcsin(R)
ATAN(R)	REAL	arctan(R)
COS(R)	REAL	cos(R)
COSH(R)	REAL	cosh(R)
DIM(IR1,IR2)	Generic	Positive difference: MAX(IR1-IR2,0)
EXP(R)	REAL	
INT(R)	INTEGER	INTEGER portion of R
LOG(R)	REAL	Natural logarithm: logeR
LOG10(R)	REAL	Common logarithm: log10R
MAX(IR1,IR2,...)	Generic	Largest of IR1,IR2,...

```
MIN(IR1,IR2,...)   Generic    Smallest of IR1,IR2,...

MOD(IR1,IR2)       Generic    Remainder: IR1-INT(IR1/IR2)*IR2

NINT(R)            INTEGER    Nearest integer: INT(ANINT(R))

REAL(I)            REAL       Real equivalent of I

SIGN(IR1,IR2)      Generic    Transfer of sign: IR1 if IR20
                               IR1 if IR2<0

SIN(R)             REAL       sin(R)

SINH(R)            REAL       sinh(R)

SQRT(R)            REAL       R

TAN(R)             REAL       tan(R)

TANH(R)            REAL       tanh(R)
```

Figure 18: Some common intrinsic functions

External functions

As well as using the intrinsic functions provided by the language, a programmer may create and use his/her own **external** functions. These functions may be included in the same source file as a program which uses them and compiled along with it, or may be written and compiled separately to obtain separate object files which are then linked to the object version of the program to obtain an executable program, in the same way as the library subprograms shown in Figure 3 on page 2. In either case, the program and functions are entirely independent **program units**.

A FORTRAN source file consists of one or more program units in any order. One of these may be a **main program unit**, which begins with an optional **PROGRAM** statement and ends with an **END** statement. The others are **subprograms,** which may be external functions or **subroutines**. (Subroutines are explained later in the chapter.)

An **external function program unit** begins with a **FUNCTION** statement and ends with an **END** statement.

Figure 19 illustrates a FORTRAN source file containing three program units, a main program MAIN and two functions FUN1 and FUN2. The order of the program units is immaterial.

```
PROGRAM MAIN
        .
        .
END
FUNCTION FUN1(arg1,...)
        .
        .
END
FUNCTION FUN2(arg1,...)
        .
```

```
        .
END
```

Figure 19: A FORTRAN source file containing two functions

Provided that the program MAIN includes no references to any other external functions, the file could be compiled, and the resulting object file linked with the library subprograms to obtain an executable program.

The functions might also be placed in one or two separate files and compiled separately from the main program. The object file or files thus obtained could then be linked with the library subprograms and the object version of the program MAIN or any other program containing references to them. In this way a programmer can create his/her own subprogram libraries for use by any program.

The FUNCTION statement

As shown above, an external function must begin with a FUNCTION statement. This has the form:

```
[type] FUNCTION name([argument_list])
```

As before, square brackets indicate that an item is optional.

Type

Each function has a type corresponding to the type of value returned by a reference to it. As for variables, the type of a function may be specified explicitly or assigned implicitly according to the first letter of the function name. For example, the function:

```
FUNCTION FUN1(arg1,...)
```

returns a value of type REAL, but

```
INTEGER FUNCTION FUN1(arg1,...)
```

returns a value of type INTEGER.

If the type of a function differs from that implied by the first letter of its name, it must be declared in a type specification in any program which refers to it. Thus any program using the second version of `FUN1` above would include the name `FUN1` in an INTEGER type specification statement, e.g.

```
INTEGER FUN1
```

The argument list

argument_list is an optional list of **dummy arguments**, separated by commas. Each dummy argument is a name similar to a variable or array name, which represents a corresponding **actual argument** used in a function reference. Dummy arguments, and variables used in a function, are

defined only within it. They may therefore be identical to variable or array names used in any other program unit.

If a dummy argument represents an array, it must appear in a type specification or DIMENSION statement in the function. If it represents a variable, it may appear in a type specification, or may be typed by default.

Example:

```
FUNCTION FUN1(A,B,N)
REAL A(100)
INTEGER B
```

Here, `A` represents a REAL array of dimension 100, and `B` and `N` represent INTEGER variables.

A function may have no arguments, e.g.

```
FUNCTION NOARGS()
```

The function reference

As we have seen, a function reference has the form:

name(*argument_list*)

argument_list is a list of **actual arguments**, which must match the list of dummy arguments in the FUNCTION statement with respect to the number of arguments and the type of each argument. For example:

```
REAL X(100)
       .
RESULT = FUN1(X,J,10)
```

would be a valid reference to the function `FUN1(A,B,N)` shown above.

If a dummy argument is a variable name, the corresponding actual argument may be any expression of the same type, i.e. a constant, variable, array element or more complex arithmetic expression.

If a dummy argument is an array name, the actual argument may be an array or array element. The dimensions of the dummy array may be variable if they are also dummy arguments.

Example:

```
REAL X(5,10)
       ...
Y = FUN(X,5,10)
       ...
END
FUNCTION FUN(A,M,N)
REAL A(M,N)
```

```
        ...
```

Actual and dummy arguments

The dummy arguments and corresponding actual arguments provide a means of exchanging information between a program unit and a function.

Each actual argument refers to a word or other unit of storage. However, no storage is reserved for a dummy argument; it is simply a name. When a function reference is evaluated, the address of each actual argument is passed to the function, and the corresponding dummy argument is set to refer to it. The dummy argument may therefore be used in the function as a variable or array referring to the same unit of storage as the actual argument.

Thus if a dummy argument represents a variable, its value on entry to the function is that of the corresponding actual argument when the function is referenced. If its value is changed in the function by an assignment or READ statement, the actual argument will be correspondingly changed after the function reference has been evaluated.

Arrays as arguments

If a dummy argument is an array, the corresponding actual argument may be an array or array element. In the former case, the elements of the dummy array correspond to the elements of the actual array in the order of their storage in memory. This, however, does not imply that the subscripts are identical, or even that the two arrays have the same number of subscripts. For example, suppose that the function:

```
FUNCTION FUN(A)
REAL A(9,6)
        .
END
```

is referenced by program MAIN as follows:

```
PROGRAM MAIN
REAL X(100),Y(0:5,-10,10)
        .
F1 = FUN(X)
F2 = FUN(Y)
        .
END
```

Then the correspondence between some elements of the dummy array `A` and the actual arrays `X` and `Y` in the two function references is as shown below:

```
A(1,1)   X(1)     Y(0,-10)

A(6,1)   X(6)     Y(5,-10)

A(7,1)   X(7)     Y(0,-9)

A(1,2)   X(10)    Y(3,-9)

A(5,4)   X(32)    Y(1,-5)
```

```
A(9,6)   X(54)    Y(5,-2)
```

If the actual argument is an array element, the first element of the dummy array corresponds to that element. Thus, if the function references:

```
F3 = FUN(X(15))
F4 = FUN(Y(3,0))
```

were included in the program above, the following items would correspond in the two references:

```
A(1,1)   X(15)    Y(3,0)

A(4,1)   X(18)    Y(0,1)

A(9,1)   X(23)    Y(5,1)

A(1,2)   X(24)    Y(0,2)

A(5,4)   X(46)    Y(4,5)

A(9,6)   X(68)    Y(2,9)
```

Such complicated relationships between actual and dummy arguments can sometimes be useful, but are in general best avoided for reasons of clarity.

Evaluation of a function

Once the dummy arguments have been initialised as described above, the statements comprising the body of the function are executed. Any statement other than a reference to the function itself may be used. At least one statement must assign a value to the function name, either by assignment, or less commonly, by a READ statement. Execution of the function is stopped, and control returned to the program unit containing the function reference, by a **RETURN** statement, written simply as:

```
RETURN
```

The value of the function name when RETURN is executed is returned as the function value to the program unit containing the function reference.

Examples

We can now write the function DGTORD suggested at the beginning of the chapter, to convert an INTEGER value representing an angle in degrees, to a REAL value representing the equivalent in radians. Our function uses the intrinsic function ATAN to compute the conversion factor.

```
FUNCTION DGTORD(DEG)
INTEGER DEG
CONFAC = ATAN(1.0)/45.0
```

```
        DGTORD = DEG*CONFAC
        RETURN
        END
```

As a second example, the following function returns the mean of an array of N real numbers.

```
        REAL FUNCTION MEAN(A,N)
        REAL A(N)
        SUM = 0.0
        DO 10, I=1,N
        10      SUM = SUM+A(I)
        MEAN = SUM/N
        RETURN
        END
```

Note that, since the type of this function differs from that implied by the first letter of its name, any program referring to it must declare the name in a type specification, e.g.

```
        REAL MEAN
```

Statement functions

If a function involves only a computation which can be written as a single statement, it may be declared as a **statement function** in any program unit which refers to it. The declaration has the form:

name(*argument_list*) = *expression*

where:

name is the name of the statement function.

argument_list is a list of dummy arguments.

expression is an expression which may include constants, variables and array elements defined in the same program unit, and function references.

The declaration must be placed after all type specifications, but before the first executable statement.

Thus the function DGTORD might be declared as a statement function in the program ANGLES:

```
        DGTORD(DEGREE) = DEGREE*ATAN(1.0)/45.0
```

Rules

The name of a statement function must be different from that of any variable or array in the same program unit.

The type of a statement function may be specified explicitly in a separate type specification or determined implicitly by the first letter of its name.

A dummy argument may have the same name as a variable or array in the same program unit. If so, it has the same type as the variable or array but is otherwise distinct from it and shares none of its attributes. For example, in the program ANGLES, the dummy argument `DEGREE` of the statement function DGTORD has the same name as the variable `DEGREE` declared in the program, and therefore has the correct (INTEGER) type, but is a different entity. If the program included the declaration:

```
INTEGER DEGREE(100)
```

the dummy argument `DEGREE` would be an INTEGER *variable*, not an array.

If a dummy argument does not have the same name as a variable or array in the same program unit, it is typed implicitly according to its first letter, e.g.

```
DGTORD(IDEG) = IDEG*ATAN(1.0)/45.0
```

expression may include references to functions, including statement functions. Any statement function must have been previously defined in the same program unit.

Subroutines

A **subroutine** is a subprogram similar in most respects to a function. Like a function, a subroutine has a list of dummy arguments used to exchange information between the subroutine and a program unit referring to it. Unlike a function, a subroutine does not return a value via its name (and therefore has no type), but it may return one or more values via its arguments.

A subroutine subprogram begins with a **`SUBROUTINE`** statement and ends with `END`. The `SUBROUTINE` statement has the form:

```
SUBROUTINE name[(argument_list)]
```

where *`name`* and *`argument_list`* have the same meanings as in the `FUNCTION` statement. The square brackets indicate that the item (*argument_list*) is optional, i.e. a subroutine may have no arguments, in which case the `SUBROUTINE` statement is simply:

```
SUBROUTINE name
```

As for a function, a subroutine must include at least one `RETURN` statement to return control to the program unit referring to it.

A subroutine is referenced by a `CALL` statement, which has the form:

```
        CALL name[(argument_list)]
```

where *argument_list* is a list of actual arguments corresponding to the dummy arguments in the SUBROUTINE statement. The rules governing the relationship between actual and dummy arguments are the same as for functions.

Functions (intrinsic and external) and subroutines are often called **procedures**.

In Example 1 of Chapter 8, the steps required to print a page header and column headers at the top of each page might be written as a subroutine. The steps are:

1. Increment the page number.
2. Print a page header, page number and column headers, followed by a blank line.

The subroutine therefore has two dummy arguments, one representing the page number and the other representing the output device, and includes the WRITE statement and FORMAT statements required to print the page and column headers. The subroutine follows:

```
        SUBROUTINE HEADER(PAGENO,OUTPUT)
C PRINT PAGE HEADER, NUMBER AND COLUMN HEADERS
        INTEGER PAGENO,OUTPUT
        PAGENO = PAGENO+1
        WRITE(OUTPUT,100)PAGENO
        100     FORMAT(1H1//1X,'DEGREES TO RADIANS CONVERSION TABLE',
        *       T74,'PAGE',I2//1X,'DEGREES  RADIANS'/)
        RETURN
        END
```

Note that the argument OUTPUT is used to receive a value from the calling program, while PAGENO both receives and returns a value.

The degrees to radians conversion program can now be rewritten using the subroutine HEADER and function DGTORD as follows:

```
        PROGRAM ANGLES
        INTEGER DEGREE,PAGENO,OUT
        DATA OUT/6/
C UNIT NUMBER FOR OUTPUT. A DIFFERENT DEVICE COULD BE USED BY
C CHANGING THIS VALUE
        DATA PAGENO/0/
        DO 10, DEGREE = 1,360
        N = DEGREE-1
        IF (N/40*40 .EQ. N) THEN
        CALL HEADER(PAGENO,OUT)
        ELSE IF (N/10*10 .EQ.N) THEN
                WRITE(OUT,110)
        END IF
        10      WRITE(OUT,120)DEGREE,DGTORD(DEGREE)
        110     FORMAT(1X)
        120     FORMAT(1X,I5,T10,F7.5)
        END
```

Figure 20: Degrees to radians conversion program (version 4)

Procedures as arguments

A program unit can pass the names of procedures as arguments to a function or subroutine. The calling program unit must declare these names in an **EXTERNAL** statement for external procedures (functions or subroutines), or **INTRINSIC** statement for intrinsic functions. The statements have the form:

EXTERNAL *list*

and INTRINSIC *list*

respectively, where *list* is a list of external procedures, or intrinsic functions respectively.

If an actual argument is a procedure name, the corresponding dummy argument may be:

1. used as a procedure in a CALL statement or function reference, or:
2. passed as an actual argument to another procedure. In this case, it must be listed in an EXTERNAL statement.

In this way, a procedure name can be passed from one procedure to another for as many levels as required.

Example 1

In Figure 21, the program MAIN passes the names of the subroutine ANALYS and the intrinsic function SQRT as actual arguments to the subroutine SUB1, corresponding to its dummy arguments SUB and FUN respectively. In SUB1, SUB appears in a CALL statement in which it is replaced in this instance by a call of ANALYS, while FUN appears in an EXTERNAL statement and is passed as an actual argument to SUB2, corresponding to its dummy argument F. In SUB2, F appears followed by a left parenthesis. Because F is not declared as an array, this is interpreted as a function reference, and is replaced by a reference to SQRT.

Note that although SQRT is an intrinsic function and is declared as such in program MAIN, FUN, the corresponding dummy argument of subroutine SUB1, is declared in SUB1 as EXTERNAL because FUN is a dummy procedure name corresponding to a function defined externally to SUB1.

```
PROGRAM MAIN
EXTERNAL ANALYS
INTRINSIC SQRT
        .
CALL SUB1(ANALYS,SQRT,A,B)
        .
END
SUBROUTINE SUB1(SUB,FUN,X,Y)
EXTERNAL FUN
        .
CALL SUB(...)
        .
CALL SUB2(FUN,X,Y)
        .
END
SUBROUTINE SUB2(F,P,Q)
```

```
        .
Q = F(P)
        .
END
```

Figure 21: Procedures as arguments

Example 2

In Figure 22, the subroutine `TRIG` has three dummy arguments, `X` representing an angle in radians, `F` representing a trigonometric function, and `Y` representing that function of `X`. The main program includes four calls to `TRIG`, using the intrinsic functions `SIN`, `COS` and `TAN` and the external function `COT`, which computes the cotangent.

```
PROGRAM MAIN
EXTERNAL COT
INTRINSIC SIN,COS,TAN
        .
CALL TRIG(ANGLE,SIN,SINE)
        .
CALL TRIG(ANGLE,COS,COSINE)
        .
CALL TRIG(ANGLE,TAN,TANGT)
        .
CALL TRIG(ANGLE,COT,COTAN)
        .
END
SUBROUTINE TRIG(X,F,Y)
Y = F(X)
RETURN
END
FUNCTION COT(X)
COT = 1.0/TAN(X)
RETURN
END
```

Figure 22: Subroutine to compute any trigonometric function.

Local variables

The variables used in a subprogram, other than its arguments, are **local variables**, defined only within it, and therefore distinct from any identically named variables used elsewhere. When a RETURN statement is executed, they become *undefined*, and their addresses may be used by other program units. Therefore, if a subprogram is executed several times, the values of its local variables are not preserved from one execution to the next.

The values of local variables can be preserved by a **SAVE** statement, which has the form:

```
SAVE [variable_list]
```

where *variable_list* is a list of local variables, separated by commas. The statement causes the values of all variables in *variable_list* to be saved. If *variable_list* is omitted, the values of all local variables are saved.

SAVE is a non-executable statement and must be placed before the first executable statement or DATA statement.

Example:

Each time the following function is executed, it prints a message indicating how many times it has been referenced.

```
FUNCTION AVE(X,Y)
INTEGER COUNT
SAVE COUNT
DATA COUNT/0/
COUNT = COUNT+1
WRITE(6,10)COUNT
       ...
10     FORMAT(1X,'FUNCTION AVE REFERENCED',I3,' TIMES.')
              ...
END
```

Control Structures - Iteration

The last chapter showed how a sequence of instructions can be executed **once**, **if** a condition is true. The need also frequently arises to execute a sequence of instructions **repeatedly**, **while** a condition is true, or **until** a condition becomes true. Such repetitive execution is called **iteration**.

:

Write a program to read the marks of a class of students in an exam, print the number of marks and compute and print the average mark. The marks are to be read one at a time, with a 'dummy' mark of 999 marking the end.

The outline of the program is:

1. Initialise the total mark to zero.
2. Initialise a count of the number of marks to zero.
3. Read the first mark.
4. While the current mark is not 999, repeat:
    1. Increment the count.
    2. Add the mark to the total.
    3. Read the next mark.
5. If the count exceeds zero then
    1. Print the count.
    2. Compute and print the average mark.

Otherwise:

1. Print message: 'No marks read.'

Unlike more modern programming languages, FORTRAN lacks a **while** structure as such, but the effect can be obtained using an IF structure and a new statement, the **GOTO**.

The GOTO statement

The GOTO statement has the form:

GOTO *label*

where *label* is the label of an executable statement, with certain restrictions which will be considered later.

A GOTO statement causes the flow of execution to 'jump' to the labelled statement and resume from there.

We can use a GOTO to complete the program of Example 1:

```
        PROGRAM AVMARK
        INTEGER TOTAL,COUNT
        TOTAL = 0
        COUNT = 0
        READ *, MARK
        10      IF (MARK.NE.999) THEN
        COUNT = COUNT+1
        TOTAL = TOTAL+MARK
        READ *, MARK
        GOTO 10
        END IF
        IF (COUNT.GT.0) THEN
        AVER = 1.0*TOTAL/COUNT
C MULTIPLY BY 1.0 TO CONVERT TO REAL AND AVOID TRUNCATION
        PRINT *, COUNT, 'MARKS WERE READ.'
        PRINT *, 'AVERAGE MARK IS', AVER
        ELSE
        PRINT *, 'NO MARKS WERE READ.'
        END IF
        END
```

Figure 8: Average mark program

Exercise:

The average mark program provides for the possibility that the data consists only of the terminator 999, but does no other checking. If the range of valid marks is 0 to 100, alter the program to check the validity of each mark, printing a suitable message if it is invalid, and print a count of any invalid marks with the results.

Count controlled loops

Any iterative structure is usually called a **loop**. As in Example 1, a loop of any kind can be constructed using an IF structure and one or more GOTO's.

Often a loop is controlled by a variable which is incremented or decremented on each iteration until a limiting value is reached, so that the number of iterations is predetermined. Such a loop is shown in Example 2.

Example 2:

Write a program to print a table of angles in degrees and their equivalents in radians, from 0 to 360 degrees, in steps of 10 degrees.

The program outline is:

1. Initialise the angle in degrees to 0.
2. While the angle does not exceed 360 degrees, repeat:
    1. Compute the equivalent in radians.
    2. Print the angle in degrees and radians.
    3. Increment the angle by 10 degrees.

and the program is:

```
        PROGRAM CONVRT
        INTEGER DEGREE
        CONFAC = 3.141593/180.0
C CONVERSION FACTOR FROM DEGREES TO RADIANS
        DEGREE = 0
        10      IF (DEGREE .LE. 360) THEN
        RADIAN = DEGREE*CONFAC
        PRINT *, DEGREE,RADIAN
        DEGREE = DEGREE + 10
        GOTO 10
        END IF
        END
```

Figure 9: Degrees to radians conversion program (version 1)

Loops of this kind occur frequently. Their essential features are:

- A **loop control variable** (DEGREE in the above example) is assigned an initial value before the first iteration.
- On each iteration, the control variable is incremented or decremented by a constant.
- Iteration stops when the value of the control variable passes a predefined limit.

FORTRAN provides for such loops with a structure called a **DO loop**, which is more concise and readable than a construction using IF and GOTO.

The Do-loop

A DO loop is a sequence of statements beginning with a **DO statement**. This has the form:

DO *label*, *var* = *e1, e2,* [,*e3*]

the square brackets indicating that ',*e3'* may be omitted.

label is the label of an executable statement sequentially following the DO statement called the **terminal statement** of the DO loop.

var is an INTEGER or REAL variable called the **loop control variable**.

e1, e2 and *e3* are arithmetic expressions (i.e. INTEGER or REAL constants, variables or more complex expressions).

The sequence of statements beginning with the statement immediately following the DO statement and ending with the terminal statement is called the **range** of the DO loop.

Execution

A DO loop is executed as follows:

1. The expressions *e1*, *e2* and *e3* are evaluated and if necessary converted to the type of *var*. If *e3* is omitted, a value of 1 is used. The resulting values are called the **parameters** of the loop. We shall call them *initial*, *limit* and *increment* respectively.

1. *initial* is assigned as a value to *var*.
    1. *var* is compared with *limit*, the test depending on the value of *increment* as follows:

Condition tested

increment > 0 *var limit*

increment < 0 *var limit*

If the condition tested is TRUE, then:

1. The range of the DO loop is executed.
2. *var* is incremented by *increment*.
3. Control returns to step 3.

Otherwise:

1. Iteration stops and execution continues with the statement following the terminal statement.

Examples:

```
DO 10, I = 1,5
```

causes the range of statements beginning with the next and ending with the statement labelled 10 to be executed 5 times.

```
DO 10, I = 0,100,5
```

causes the range to be executed 21 times for values of `I` of 0,5,10...100.

```
DO 10, I = 100,0,-5
```

causes the range to be executed 21 times for values of `I` of 100,95...0.

```
DO 10, I = 0,100,-5
```

In this case, the range is not executed at all, as the test in step 3 fails for the initial value of `I`.

```
DO 10, J = I,4*N**2-1,K
```

Here, *e1*, *e2* and *e3* are more complex expressions.

We can now rewrite the program of Example 2 using a DO loop. The outline becomes:

1. Repeat for angles in degrees from 0 to 360 in steps of 10:
    1. Compute the equivalent in radians.
    2. Print the angle in degrees and radians.

and the program follows:

```
        PROGRAM CONVRT
        INTEGER DEGREE
        CONFAC = 3.141593/180.0
C CONVERSION FACTOR FROM DEGREES TO RADIANS
        DO 10, DEGREE = 0,360,10
        RADIAN = DEGREE*CONFAC
        10      PRINT *, DEGREE,RADIAN
        END
```

Figure 10: Degrees to radians conversion program (version 2)

This is clearer and more concise than version 1. Note the use of indentation to clarify the loop structure.

Restrictions and other notes

To protect the integrity of the loop structure, there are various restrictions affecting DO loops.

Increment must not be zero.

The terminal statement must be one which is self-contained and allows execution to continue at the next statement. This rules out STOP, END and another DO statement. It is often convenient to end a DO loop with a **CONTINUE** statement, which has no effect whatever, serving only to mark the end of the loop.

The range of a DO loop can be entered only via the initial DO statement. Thus a GOTO cannot cause a jump into the range of a DO loop. However, GOTOs can be included in the range to jump to statements either inside or outside it. In the latter case, this can cause iteration to stop before the control variable reaches the limiting value.

Examples:

```
        GOTO 10
               .
        DO 20, I = 1,5
               .
        10             .
               .
        20     CONTINUE
```

is wrong, but

```
        DO 20, I = 1,5
               .
        10             .
                       .
        IF (...) GOTO 10
               .
        IF (...) GOTO 30
               .
        20     CONTINUE
```

```
        .
30      .
```

is all right.

The control variable can be freely used in expressions in the range of the loop (as in Figure 10) but it cannot be assigned a value.

The loop parameters are the *values* of the expressions *e1, e2* and *e3* on entry to the loop. The expressions themselves are not used. Therefore if any of *e1, e2* and *e3* are variables, they can be assigned values within the loop without disrupting its execution.

The control variable

As explained under 'Execution' the control variable is incremented and tested at the end of each iteration. Thus, unless iteration is interrupted by a GOTO, the value of the control variable after execution of the loop will be the value which it was assigned at the end of the final iteration. For example, in a loop controlled by the statement:

```
DO 10, I = 0,100,5
```

the control variable `I` is incremented to exactly 100 at the end of the 20th iteration. This does not exceed *limit*, so another iteration is performed. `I` is then incremented to 105 and iteration stops, with `I` retaining this value.

If the control variable is REAL, inconsistent results may be obtained unless allowance is made for approximation. For example, in a loop controlled by:

```
DO 10, C = 0,100,5
```

the control variable `C` is incremented at the end of the 20th iteration to a value of *approximately* 100. If it is less, execution continues for a further iteration, but if it is greater, iteration stops.

To avoid such effects, a higher value of *limit* should be used, e.g.

```
DO 10, C = 0,101,5
```

Nested DO loops

DO loops, like IF structures, can be nested, provided that there is no overlapping. (i.e. that the range of each nested loop is entirely within the range of any loop in which it is nested).

Example:

```
Valid           Invalid

DO 20 ...       DO 20 ...

.               .
```

```
 DO 10 ...          DO 10 ...

.                  .

 10 CONTINUE       20 CONTINUE

.                  .

20 CONTINUE       10 CONTINUE
```

The following provides a simple, if not very useful example of a nested loop structure.

Example 3:

Write a program to print a set of multiplication tables from 2 times up to 12 times.

The outline is:

1. Repeat for `I` increasing from 2 to 12:
    1. Print table heading.
    2. Repeat for `J` increasing from 1 to 12:
        1. Print `I` 'times' `J` 'is' `I*J`

and the program is:

```
PROGRAM TABLES
DO 20, I = 2,12
PRINT *,I,' TIMES TABLE'
DO 10, J = 1,12
10      PRINT *,I,' TIMES',J,' IS',I*J
20      CONTINUE
END
```

Figure 11: Multiplication tables program

There is no logical need for the CONTINUE statement in this program as nested loops can share a common terminal statement. Thus the program could be rewritten as:

```
PROGRAM TABLES
DO 10, I = 2,12
PRINT *,I,' TIMES TABLE'
DO 10, J = 1,12
10      PRINT *,I,' TIMES',J,' IS',I*J
END
```

However, to clarify the structure, it is better to use separate terminal statements and indentation as in the first version.

Control Structures - Conditional Execution

The FORTRAN statements covered so far are enough to allow us to read information, evaluate arithmetic expressions and print results. It is hardly necessary to write a program to perform such tasks, which can usually be more easily done using a calculator.

The main advantages of a computer are its ability to:

- execute alternative sequences of instructions depending on a condition (**conditional execution**).
- execute a sequence of instructions repeatedly while or until a condition is satisfied (**iteration**).

This chapter deals with conditional execution while iteration is covered in Chapter 6.

The need for conditional execution is illustrated by the following problem:

Example 1:

Write a program to read the coefficients of a quadratic equation and print its roots.

Solution: The roots of the quadratic equation

$$ax^2 + bx + c$$

are given by the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The outline of the program is:

1. Read the coefficients.
2. Evaluate $b^2 - 4ac$.
3. If $b^2 - 4ac$ exceeds zero then
    1. Compute and print two distinct real roots.

Otherwise, if $b^2 - 4ac$ is equal to zero then

1. Compute and print two coincident real roots.

Otherwise

1. Print message: 'No real roots.'

In step 3, the program must test conditions such as '$b^2 - 4ac$ exceeds zero'. To express such conditions, FORTRAN uses another type, the **LOGICAL** type.

The LOGICAL type

There are two **LOGICAL constants**, defined as **.TRUE.** and **.FALSE.**.

A **LOGICAL variable** can be assigned either of these values. It may not be assigned a value of any other type. Each LOGICAL variable must be declared in a LOGICAL type specification statement, which must occur, like all other type specifications, before the first executable statement.

Example:

The LOGICAL variable **ERROR** could be declared and initialised by the statements:

```
LOGICAL ERROR
ERROR = .FALSE.
```

Logical expressions

A logical expresssion is one which evaluates to one of the LOGICAL constants .TRUE. or .FALSE.. Thus the simplest logical expressions are the LOGICAL constants themselves, and LOGICAL variables.

Relational expressions

A **relational expression** is a logical expression which states a relationship between two expressions, evaluating to .TRUE. if the relationship applies or .FALSE. otherwise. For the present, we shall consider only relationships between arithmetic expressions. (As we shall see later, FORTRAN can also deal with relationships between CHARACTER expressions.)

A relational expression has the form:

arithmetic_expression relational_operator arithmetic_expression

The relational operators are:

	Meaning
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GE.	Greater than or equal to

```
.GT.   Greater than
```

Thus examples of relational expressions are:

```
N.GE.0
X.LT.Y
B**2 - 4*A*C .GT. 0.
```

Notes:

1. Relational operators have lower precedence than arithmetic operators. Therefore, in evaluating a relational expression, the arithmetic expressions are evaluated before the comparison indicated by the relational operator is made.
2. The two arithmetic expressions may be of different type (i.e. one INTEGER and one REAL). In this case, the INTEGER expression is converted to REAL form before the comparison is made.

Composite logical expressions

It is often necessary to express a condition which combines two or more logical expressions. For example, to check that the value of a variable lies within a given range, we should have to check that it is greater than the lower limit AND less than the upper limit. Such conditions are expressed in FORTRAN by **composite logical expressions**, which have the form:

L1 logical_operator L2

where *L1* and *L2* are logical expressions (relational or composite). The logical operators and their meanings are shown below. The second column indicates the conditions under which a composite logical expression as above evaluates to .TRUE..

	Meaning
.AND.	Both *L1* and *L2* are .TRUE.
.OR.	Either *L1* or *L2* or both are .TRUE.
.EQV.	Both *L1* and *L2* have the same value (.TRUE. or .FALSE.)
.NEQV.	*L1* and *L2* have different values (one .TRUE. and one .FALSE.)

Thus the following composite logical expression would evaluate to .TRUE if the value of the variable X lay within a range with non-inclusive limits MIN and MAX.:

```
X.GT.MIN .AND. X.LT.MAX
```

There is one further logical operator **.NOT.**, which unlike the others, takes only one operand, which it precedes. The expression **.NOT.*L*** is .TRUE. if the logical expression *L* is .FALSE. and vice versa.

As with arithmetic operators, precedence rules are required to define the interpretation of expressions like:

.NOT. *L1* .OR. *L2*

which could evaluate to .TRUE. under either of the following conditions, depending on the order of evaluation:

1. *L1* is .FALSE. or *L2* is .TRUE.
2. *L1* and *L2* are both .FALSE.

The precedence order is shown by the following list, in which precedence decreases downwards.

arithmetic operators

relational operators

.NOT.

.AND.

.OR.

.EQV. and .NEQV.

Thus (i) is the correct interpretation of the above expression.

As in arithmetic expressions, parentheses can be used to group partial logical expressions and change the order of evaluation. Thus

.NOT.(*L1*.OR.*L2*)

would be evaluated according to interpretation (ii).

Parentheses can also be used to improve clarity, even when not logically required, e.g.

(A.LT.B) .OR. (C.LT.D)

Logical assignment

The value of a logical expression can be assigned to a variable of type LOGICAL, e.g.

```
LOGICAL VALID
        ...
VALID = X.GT.MIN .AND. X.LT.MAX
```

Logical expressions are more commonly used in logical **IF statements** and **structures**.

The logical IF statement

The logical IF statement is used to execute an instruction conditionally. It has the form:

IF (*logical_expression) executable_statement*

where *executable_statement* is an executable FORTRAN statement other than another IF statement or a DO statement (see Chapter 6).

The statement is executed by evaluating *logical_expression* and executing *executable_statement* if it evaluates to .TRUE..

Example: `IF (A.LT.B) SUM = SUM + A`

The block IF structure

The logical IF statement is of limited usefulness, as it permits only the execution of a single instruction depending on a single condition. The **block IF structure** is more powerful, permitting the conditional execution of one of a number of alternative **sequences** of instructions. It may be described informally as:

- an **IF** block, followed by:

- one or more optional **ELSE IF** blocks, followed by:
- an optional **ELSE** block, followed by:
    - **END IF**

More formally, the structure is:

IF () THEN

ELSE IF () THEN

...

ELSE

END IF

where:

- and are logical expressions.
- , and are sequences of FORTRAN statements.
- The square brackets ([]) indicate that an item is optional and the ellipsis (...) that it may be repeated indefinitely.

The structure is executed as follows:

is evaluated. If it evaluates to .TRUE., the sequence is executed and execution continues with the statement following END IF.

Otherwise:

1. If there are any ELSE IF clauses, each is evaluated, until *either:*
    1. An evaluates to .TRUE.. The sequence is executed and execution continues with the statement following END IF.

or:

1. The last evaluates to .FALSE.. Execution continues with step 2.

1. If there is an ELSE clause, the sequence is executed.
2. Execution continues with the statement following END IF.

Thus, a simple block IF structure is:

```
IF (A.LT.B) THEN
SUM = SUM + A
PRINT *, SUM
END IF
```

which is equivalent to the IF statement shown earlier.

A more realistic example is the following:

Example:

An employee is paid at the standard rate for the first 40 hours of work, at time and a half for the next 10, and at double time for any hours in excess of 50. If the variable HRS represents the hours worked and RATE the standard rate then the employee's salary is computed by the block IF structure:

```
IF (HRS.LE.40) THEN
        SALARY = HRS*RATE
ELSE IF (HRS.LE.50) THEN
        SALARY = 40.0*RATE + (HRS-40.0)*RATE*1.5
ELSE
        SALARY = 40.0*RATE + 10.0*RATE*1.5 + (HRS-50.0)*RATE*2.0
END IF
```

Note the use of indentation to clarify the structure.

We are now in a position to complete the quadratic roots program of Example 1, but first the outline should be altered as follows:

1. Because REAL values are approximations, exact comparisons involving them are unreliable. The relational expressions in our program should therefore be reformulated using a variable (for error) to which a small positive value has previously been assigned. The expression ' exceeds zero' in step 3 should be replaced by:

>

Similarly, the expression ' is equal to zero' in step 3 should be replaced by:

However, the expression is evaluated only if > has previously been evaluated as false, which of course implies . Therefore, all that is required is:

Comparisons involving REAL values should always be expressed in this way.

1. The expression for the roots includes a divisor of . Therefore if has a value of zero, the evaluation of this expression will cause the program to fail with an arithmetic error. The program should prevent this by testing and printing a suitable message if it is zero. Again, the test should be expressed using the error variable .

In general, programs should be designed to be **robust**, i.e. they should take account of any exceptional data values which may cause the program to fail, and take steps to prevent this.

The program outline now becomes:

1. Assign a small positive value to .

1. Read the coefficients.
2. If then
    1. Print message: 'First coefficient must be non-zero'.

Otherwise:

1. Evaluate
2. If > then
    1. Compute and print two distinct real roots.

Otherwise, if then

1. Compute and print two coincident real roots.

Otherwise

1. Print message: 'No real roots.'

Now that the outline is complete, the program can be easily written:

```
PROGRAM QUAD
E = 1E-9
READ *, A,B,C
IF (A.GE. -E .AND. A.LE.E) THEN
PRINT *, 'FIRST COEFFICIENT MUST BE NON-ZERO.'
ELSE
S = B**2 - 4*A*C
IF (S.GT.E) THEN
D = S**0.5
X1 = (-B+D)/(2*A)
X2 = (-B-D)/(2*A)
```

```
PRINT *, 'TWO DISTINCT ROOTS:' X1 'AND' X2
ELSE IF (S.GT. -E) THEN
X = -B/(2*A)
PRINT *, 'TWO COINCIDENT ROOTS',X
ELSE
PRINT *, 'NO REAL ROOTS.'
END IF
END IF
END
```

Figure 7: Quadratic roots program

Note that most of the program consists of a block IF structure, with a second block IF included in its ELSE clause. The embedding of one structure within another in this way is called **nesting**.

Once again, indentation has been used to clarify the structure.

Input and output

This chapter introduces input, output and format statements which give us greater flexibility than the simple READ and PRINT statements used so far.

A statement which reads information must:

1. Scan a stream of information from an input device or file.
2. Split the stream of information into separate items.
3. Convert each item from its **external** form in the input to its **internal** (binary) representation.
4. Store each item in a variable.

A statement which outputs information must:

1. Retrieve each item from a variable or specify it directly as a constant.
2. Convert each item from its internal form to an external form suitable for output to a given device or file.
3. Combine the items with information required to control horizontal and vertical spacing.
4. Send the information to the appropriate device or file.

The simple READ statement:

`READ` *, *variable_list*

reads a line (or **record**) of information from the **standard input** (defined as the keyboard for programs run from a terminal) and stores it in the variables in *variable_list*. The asterisk refers to a **list-directed format** used to split the information into separate items using spaces and/or commas as separators and convert each item to the appropriate internal representation, which is determined by the type of the corresponding variable in *variable_list*.

Similarly, the simple PRINT statement:

PRINT *, *output_list*

uses a list-directed format to convert each constant, and the value of each variable, in *output_list* to a suitable form for output on **standard output** (defined for a program run from a terminal as the screen) and prints the list as a line of output, with spaces between the items.

The FORMAT statement

We can obtain greater control over the conversion and formatting of input/output items by replacing the asterisk in a READ or PRINT statement by the label of a **FORMAT** statement, for example:

```
READ 10,A,B,C
```

```
        .
10      FORMAT(...)
```

The FORMAT statement describes the layout of each item to be read or printed, and how it is to be converted from external to internal form or vice versa. It also describes the movements of an imaginary *cursor* which can be envisaged as scanning the input list. Its general form is:

label `FORMAT` (*specification_list*)

label is a statement label. A FORMAT statement must always be labelled to provide a reference for use in input/output statements.

specification_list is a list of **format descriptors** (sometimes called **edit descriptors**), separated by commas. These describe the layout of each input or output item, and specify how it is to be converted (or edited) from external to internal form or vice versa.

FORMAT statements can be placed anywhere in a program. It is often convenient to place them all at the end (immediately before END), especially if some of them are used by more than one input/output statement.

Formatted input

The format descriptors used for input are summarised in Figure 14 and described in the following sections.

Descriptor	Meaning
I*w*	Convert the next *w* characters to an INTEGER value.
F*w.d*	Convert the next *w* characters to a REAL value. If no decimal point is included, the final *d* digits are the fractional part.
E*w.d*	Convert the next *w* characters to a REAL value, interpreting them as a number in exponential notation.
*n***X**	Skip the next *n* characters.
T*c*	Skip to character position *c*.
TL*n*	Skip to the character *n* characters to the left of the current character.
TR*n*	Skip to the character *n* characters to the right of the current character.

Figure 14: Some format descriptors for input

The I format descriptor

This is used to read a value into an INTEGER variable. Its form is **I***w*, where *w* is an unsigned integer indicating the number of characters to be read (the *width* of the *field*). These characters must consist of decimal digits and/or spaces, which are interpreted as zeroes, with an optional + or - sign anywhere before the first digit. Any other characters will cause an input error.

Example:

FORTRAN: `READ 10,MEAN,INC`

```
10      FORMAT(I4,I4)
```

Input: *b*123*b*-5*b*

(*b* represents a blank). This assigns a value of 123 to `MEAN` and -50 to `INC`.

The F format descriptor

This is used to read a value into a REAL variable. It has the form **F***w.d*, where *w* is an unsigned integer representing the width of the field and *d* is an unsigned integer representing the number of digits in the fractional part.

The corresponding input item must consist of decimal digits and/or spaces, with an optional sign anywhere before the first digit and an optional decimal point. As with the `I` format descriptor, spaces are interpreted as zeroes. If there is no decimal point in the item, the number of fractional digits is indicated by *d*. If the item includes a decimal point, *d* is ignored, and the number of fractional digits is as indicated.

Example:

FORTRAN: `READ 10,X,A,B,C,D`

```
10      FORMAT(F4.5,F4.1,F2.2,F3.5,F3.0)
```

Input: *b*1.5*b*123456789*bb*

Results: `X`: 1.5 `A`: 12.3 `B`: 0.45 `C`: 0.00678 `D`: 900.0

The E format descriptor

This is used to read a value into a REAL variable. It has a similar form to the F format descriptor, but is more versatile, as it can be used to read input in exponential notation.

We saw in Chapter Two that a REAL constant can be written in exponential notation as a REAL or INTEGER constant followed by an exponent in the form of the letter 'E' followed by the power of 10 by which the number is to be multiplied. For input, the exponent can also be a signed integer without the letter 'E'.

Example:

With a format descriptor of E9.2, all the following will be read as 1.26

0.126E*b*01

1.26*b*E*b*00

1.26*bbbbb*

12.60E-01

bbb.126E1

*bbbbbb*126

126bbbbbb

bbb12.6-1

Repeat count

The I, F and E format descriptors may be repeated by preceding them by a number indicating the number of repetitions. For example:

```
10 FORMAT(3I4)
```

is equivalent to:

```
10 FORMAT(I4,I4,I4)
```

The X format descriptor

This is used with an unsigned integer prefix *n* to skip *n* characters.

Example:

FORTRAN: `READ 10,I,J`

```
10      FORMAT(I4,3X,I3)
```

Input: 123456789*b*

Results: `I`: 1234 `J`: 890

The T format descriptors

The T (for **tab**), TL and TR format descriptors are used to move the cursor to a given position. This is defined *absolutely* by the T format descriptor or *relative* to the current position by the TL and TR descriptors.

Example:

FORTRAN: `READ 10,I,J,K`

```
      10     FORMAT(T4,I2,TR2,I2,TL5,I3)
```

Input: 123456789*b*

Results: `I`: 45 `J`: 89 `K`: 567

Notes:

1. TR*n* is equivalent to *n*X.
2. As illustrated by the example, tabs can be used not only to skip over parts of the input, but to go back and re-read parts of it.
3. If TL*n* defines a position before the start of the record, the cursor is positioned at the first character. TL with a large value of *n* can therefore be used to return the cursor to the beginning of the record (as can T1).

Formatted output

Output statements use the same format descriptors as for input and another, the **literal** format descriptor, which is a string of characters for output. The descriptors are summarised in Figure 15 and described further in the following sections.

Descriptor	**Meaning**
I*w*	Output an INTEGER value in the next *w* character positions
F*w.d*	Output a REAL value in the next *w* character positions, with *d* digits in the fractional part.
E*w.d*	Output a REAL value in exponential notation in the next *w* character positions, with *d* digits in the fractional part.
*n***X**	Skip the next *n* character positions.
T*c*	Skip to character position *c*.
TL*n*	Skip to the character *n* characters to the left of the current character.
TR*n*	Skip to the character *n* characters to the right of the current character.
'*c1c2...cn* '	Output the string of *n* characters *c1c2...cn* starting at the next character position.
*n***H***c1c2...c**n*	Output the string of *n* characters *c1c2...cn* starting at the next character position.

Figure 15: Some format descriptors for output

Vertical spacing

As well as defining the layout of a line of output via an associated FORMAT statement, an output statement must define the vertical placement of the line on the screen or page of printed output. The method of doing this is described before the use of the format descriptors of Figure 15.

The computer uses the output list and the corresponding format specification list to build each line of output in a storage unit called an **output buffer** before displaying or printing it. When the contents of the buffer are displayed on the screen or printed on paper, the first character is not shown, but is interpreted as a **control character**, defining the vertical placement of the line. Four control characters are recognised, as shown in Figure 16.

Character	**Vertical spacing before output**
Space	One line
0 (zero)	Two lines
1	New page
+	No vertical spacing (i.e. Current line is overprinted).

Figure 16: Control characters for vertical spacing

The effect of any other character is not defined, but is usually the same as a space, i.e. output is on the next line.

Incorrect output may be obtained if the control character is not taken into account. It is therefore best to use the format specification to insert a control character as the first character in a line, rather than to provide it via the output list. For example:

```
N = 15
PRINT 10,N
10      FORMAT(1X,I2)
```

Buffer contents: *b*15

Output: 15

The initial blank in the buffer is interpreted as a control character, and '15' is printed on the next line. However, if the FORMAT statement were:

```
10      FORMAT(I2)
```

the buffer contents would be '15'. On printing, the initial '1' would be interpreted as a control character, and '5' would be printed at the start of the next page.

The following sections describe in more detail the effect of the format descriptors in output statements.

The I format descriptor

The format descriptor `I`*w* is used to print an INTEGER value right-justified in a field of width *w* character positions, filling unused positions on the left with blanks and beginning with a '-' sign if the value is negative. If the value cannot be printed in a field of width *w,* the field is filled with asterisks and an output error is reported.

Example:

```
I = 15
J = 709
K = -12
PRINT 10,I,J,K,
10      FORMAT(1X,I4,I4,I4)
```

Output: *bb*15*b*709*b-12*

Notes:

1. The first format descriptor `1X` provides a space as a control character to begin output on a new line. The next descriptor `I4` then prints the value 15 in a field of width 4. The same effect could be obtained by using `I5` as the first descriptor, but it is clearer to use a separate descriptor for the control character.
2. The `I`, `F` and `E` format descriptors may be preceded by a **repetition count** *r*, where *r* is an unsigned integer. Thus *r*`I`*w* repeats the format descriptor `I`*w* for *r* repetitions. For example, the above FORMAT statement could be replaced by:

```
10      FORMAT(1X,3I4)
```

The F format descriptor

The format descriptor `F`*w.d* (F for floating point) is used to print a REAL value right-justified in a field of width *w*, with the fractional part *rounded* (not truncated) to *d* places of decimals. The field is filled on the left with blanks and the first non-blank character is '-' if the value is negative. If the value cannot be printed according to the descriptor, the field is filled with asterisks and an error is reported.

Example:

```
X = 3.14159
Y = -275.3024
Z = 12.9999
PRINT 10,X,Y,Z,
10      FORMAT(1X,3F10.3)
```

Output: *bbbbb*3.142*bb*-275.302*bbbb*13.000

The value of X is rounded up, that of Y is rounded down, and that of Z is rounded up, the 3 decimal places being filled with zeroes.

The E format descriptor

The format descriptor `E`*w.d* is used to print a REAL value in exponential notation right-justified in a field of width *w*, with the fractional part rounded to *d* places of decimals. Thus the layout for a format descriptor of `E10.3` is:

*S*0.*XXX*E*SXX*

<*d*>

<-----*w*------>

S indicates a position for a sign. The initial sign is printed only if negative, but the sign of the exponent is always printed.

X indicates a digit.

Example:

The value 0.0000231436 is printed as shown with the various format descriptors:

E10.4 0.2314E-04

E12.3 *bbb*0.231E-04

E12.5 *b*0.23144E-04

The literal format descriptors

The literal format descriptors '*c1c2...cn*' and *n***H***c1c2...cn* place the string of *n* characters *c1c2...cn* directly into the buffer. Thus a PRINT statement using either of the following FORMAT statements will print the header: 'RESULTS' at the top of a new page:

```
10 FORMAT('1','RESULTS')
10 FORMAT(1H1,7HRESULTS)
```

The quoted form is generally easier to use, but the 'H' form is convenient for providing control characters.

A repetition count may be used with a literal format descriptor if the descriptor is enclosed in parentheses, e.g.

```
10 FORMAT(1X,3('RESULTS'))
```

More general input/output statements

A **record** is a sequence of values or characters.

A **file** is a sequence of records.

An **external file** is one contained on an external medium (e.g. a magnetic disk).

Each FORTRAN input/output statement reads information from, or writes it to, a file. The file must be **connected** to an **external unit**, i.e. a physical device such as the keyboard or screen, or a magnetic disk. An external unit is referred to by a **unit identifier**, which may be:

- a non-negative integer expression, or:
- an asterisk ('*'), which normally refers to the keyboard for input, or the screen for output.

The READ and PRINT statements we have considered so far read from the file 'standard input', normally connected to the keyboard, and print on the file 'standard output', normally connected to the screen. To use different files and devices and to obtain various other options, we require a more general form of the READ statement for input, and a new statement, the WRITE statement for output. These statements have the form:

`READ` (*cilist*) *input_list*

and `WRITE` (*cilist*) *output_list*

where *cilist* is a list of **input-output specifiers**, separated by commas. Each specifier takes the form:

keyword = value

The specifiers may be in any order. In special cases noted below, only the value is required. Some of the keywords are:

UNIT

FMT

ERR

END

The unit specifier must always be included. Its value must be a unit identifier, as defined above.

If the unit specifier is the first item in *cilist*, it may be denoted by its value only (without 'UNIT=').

Unit identifiers 5 and 6 are preconnected to the files 'standard input' and 'standard output' respectively.

The value of the format specifier FMT is the label of a FORMAT statement to be used for input/output conversion, or an asterisk to indicate list-directed formatting. A format specifier may be denoted by its value only (without 'FMT=') if it is the second item in *cilist* and follows a unit specifier also denoted by its value only.

Examples:

If unit identifier 5 corresponds to standard input, the following are all equivalent:

```
READ(UNIT=5,FMT=100) X,Y,Z
READ(FMT=100,UNIT=5) X,Y,Z
READ(5,FMT=100) X,Y,Z
READ(5,100) X,Y,Z
READ(*,100) X,Y,Z
```

Also, the statements:

```
        READ(*,*) A,B,C
and     READ(5,*) A,B,C
```

are both equivalent to the list-directed input statement:

```
READ *, A,B,C
```

The last two specifiers deal with special conditions. If an error occurs in input or output execution normally stops, but if an error specifier of the form:

`ERR` = *label*

is included in *cilist*, execution continues from the statement labelled *label*. This makes it possible to include statements in the program to take special actions to deal with such errors.

If a READ statement tries to read more data than is available, an input error normally occurs. However, if a file ends with a special **end-of-file** record, a specifier of the form:

`END` = *label*

will cause execution to continue from the statement labelled *label*.

The OPEN statement

As noted above, the files 'standard input' and 'standard output' are preconnected to unit identifiers 5 and 6, and normally refer to the keyboard and screen, respectively. If other files, e.g. files on disk, are to be used, or if 'standard input' and 'standard output' are to be redefined, each file must be connected to an external unit by an **OPEN** statement, which has the form:

```
OPEN(openlist)
```

where *openlist* is a list of specifiers of the form:

keyword = *value*

Specifiers may occur in any order. Two of the more important keywords are:

UNIT

FILE

The unit specifier must be included. Its value must be a unit identifier.

If the unit specifier is the first item in *openlist*, it may be denoted by its value only (without 'UNIT=').

The value of the file specifier is a character expression naming a file to be opened, i.e. connected to the external unit specified by the unit specifier. If the file does not exist, a new file is created.

Example:

```
OPEN(8, FILE='MYFILE.DAT')
```

connects the file MYFILE.DAT to unit 8. If the file does not exist, it is created. READ and WRITE statements referring to this unit identifier will then read from or write to this file.

Repetition of format specifications

If the number of items in an input or output list exceeds the number of format descriptors in the corresponding FORMAT statement, a new record is taken (a new line for terminal input/output) and the format specification list is re-used. This happens as often as required to deal with the complete list.

Example:

```
READ(5,10) A,B,C,P,Q,R,X,Y,Z
10      FORMAT(3F12.3)
```

This reads three values from the first line of input into the variables `A,B` and `C`, from the second line into `P,Q` and `R`, and from the third line into `X,Y` and `Z`. Similarly:

```
WRITE(6,10) A,B,C,P,Q,R,X,Y,Z
10      FORMAT(1X,3F12.3)
```

prints the values of the variables three to a line on consecutive lines.

The format specification list may also be re-used partially if it includes nested parentheses. The rules are:

- If there are no nested parentheses, the specification list is re-used from the beginning.
- If the list includes nested parentheses, the list is re-used from the left parenthesis corresponding to the last nested right parenthesis.
- If the left parenthesis so defined is preceded by a repeat count the list is re-used from immediately before the repeat count.

This is illustrated by the following examples, in which an arrow indicates the point from which repetition, if required, begins:

```
10      FORMAT(I6,10X,I5,3F10.2)

20      FORMAT(I6,10X,I5,(3F10.2))

30      FORMAT(I6,(10X,I5),3F10.2)

40      FORMAT(F6.2,(2F4.1,2X,I4,4(I7,F7.2)))

50      FORMAT(F6.2,2(2F4.1,2X,I4),4(I7,F7.2))

60      FORMAT(F6.2,(2(2F4.1,2X,I4),4(I7,F7.2)))
```

Multi-record specifications

Repetitions can be used as in the last section to read in or print out a sequence of values on consecutive lines using the same format specification list. It is also useful to be able to specify the format of several consecutive lines (or records) in a single format specification. This can be done using the / format descriptor, which marks the end of a record. Unlike other format descriptors, / need not be preceded or followed by a comma.

On input, / causes the rest of the current record to be skipped, and the next value to be read from the first item on the next record. For example:

```
READ(5,100) A,B,C,I,J,K
100     FORMAT(2F10.2,F12.3/I6,2I10)
```

reads three REAL values into `A,B` and `C` from a line, ignores anything more on that line, and reads three INTEGER values into `I,J` and `K` from the next line.

Consecutive slashes cause records to be skipped. Thus if the FORMAT statement in the above example were changed to:

```
100     FORMAT(2F10.2,F12.3//I6,2I10)
```

a complete line would be skipped before the values were read into `I,J` and `K`.

On output, a / marks the end of a record, and starts a new one. Consecutive slashes cause blank records to be output. For example:

```
WRITE(6,200) A,B,A+B,A*B
200             FORMAT(1H1////T10,'MULTIPLE LINES EXAMPLE'///
       *        1X,'THE SUM OF',F5.2,' AND',F5.2,' IS',F5.2/
*      1X,'AND THEIR PRODUCT IS',F8.2////)
```

prints four blank lines and a header at the top of a new page, followed by two blank lines, then the sum and product on consecutive lines followed by four blank lines.

The following example illustrates the use of formatting to produce output in tabular form with headers and regular spacing.

Example 1:

Rewrite the degrees to radians conversion program (Chapter 6, Example 2) to print angles from 1 to 360 degrees in 1 degree intervals and their equivalents in radians. The results should be printed 40 lines to a page, with the values suitably formatted, blank lines separating groups of 10 consecutive lines, headers for the 'Degrees' and 'Radians' columns, and a header and page number at the start of each page.

The outline is:

1. Compute the conversion factor.
2. Initialise the page number to zero.
3. Repeat for angles in degrees from 1 to 360 in steps of 1:
    1. If the angle in degrees is one more than a multiple of 40 then:
        1. Increment the page number.
        2. Print a page header, page number and column headers, followed by a blank line.

Otherwise, if the angle in degrees is one more than a multiple of 10, then:

1. Print a blank line.

1. Compute the angle in radians.
2. Print the angle in degrees and radians.

and the program follows:

```
        PROGRAM ANGLES
        INTEGER DEGREE,PAGENO,OUT
        DATA OUT/6/
C UNIT NUMBER FOR OUTPUT. A DIFFERENT DEVICE COULD BE USED BY
C CHANGING THIS VALUE
        DATA PAGENO/0/
        CONFAC = 3.141593/180.0
C CONVERSION FACTOR FROM DEGREES TO RADIANS
        DO 10, DEGREE = 1,360
        N = DEGREE-1
        IF (N/40*40 .EQ. N) THEN
C       PRINT PAGE HEADER, NUMBER AND COLUMN HEADERS
                PAGENO = PAGENO+1
                WRITE(OUT,100)PAGENO
        ELSE IF (N/10*10 .EQ.N) THEN
                WRITE(OUT,110)
        END IF
        RADIAN = DEGREE*CONFAC
        10      WRITE(OUT,120)DEGREE,RADIAN
        100     FORMAT(1H1//1X,'DEGREES TO RADIANS CONVERSION TABLE',
        *       T74,'PAGE',I2//1X,'DEGREES  RADIANS'/)
        110     FORMAT(1X)
        120     FORMAT(1X,I5,T10,F7.5)
        END
```

Figure 17: Degrees to radians conversion program (version 3)

A Simple Program

We are now ready to write a simple FORTRAN program. All that is required is some information on printing output, program layout and a few simple statements.

The PRINT statement

Output can be printed using the PRINT statement, which is very similar to the READ statement shown on page 4:

PRINT *, *output_list*

where *output_list* is a single constant, variable or expression or a list of such items, separated by commas.

Example: `PRINT *,'THE RESULTS ARE', X ,'AND',Y`

The PRINT statement prints the output list on the terminal screen in a standard format. Later, we shall consider more flexible output statements which give us greater control over the appearance of the output and the device where it is printed.

The PROGRAM statement

A program can optionally be given a name by beginning it with a single **PROGRAM** statement. This has the form:

PROGRAM *program_name*

where *program_name* is a name conforming to the rules for FORTRAN variables. (see 'Variables' on page 3).

END and STOP

Each program must conclude with the statement **END**, which marks the end of the program. There must be no previous END statement.

The statement **STOP** stops execution of the program. In FORTRAN 77, but not in previous versions, END also has this effect. Therefore, if execution is simply to stop at the end of the program, STOP is optional. However, one or more STOP statements may be written earlier, to stop execution conditionally at points other than the end.

Program layout

When FORTRAN was introduced, punched cards were a common input medium. FORTRAN was designed to make use of the cards' 80-column layout by ignoring spaces and reserving different fields of the card for different purposes. Although cards are no longer used, FORTRAN still uses this column-based layout.

All FORTRAN statements must be written in columns 7 to 72. A statement ends with the last character on the line, unless the next line has any character other than 0 in column 6. Any such character indicates that columns 7 to 72 are a continuation of the previous line.

Columns 73 to 80 are ignored by the compiler. Originally, these columns were used to print sequence numbers on the cards, but now they are normally unused.

Columns 1 to 5 are reserved for **statement labels**. These are optional unique unsigned non-zero integers used to provide a reference to statements.

The layout rules are summarised in Figure 5.

Columns	Usage
1-5	Statement labels
6	Continuation character or blank
7-72	FORTRAN statements
73-80	Unused

Figure 5: FORTRAN layout

Comments

The letter 'C' or an asterisk in column one causes the compiler to ignore the rest of the line, which may therefore be used as a **comment** to provide information for anyone reading the program.

A simple program

The following example uses the FORTRAN statements introduced so far to solve a simple problem.

Example 1:

A driver fills his tank with petrol before setting out on a journey. Each time he stops for petrol he puts in 40 litres. At his destination, he fills the tank again and notes the distance he has travelled in kilometres. Write a program which reads the distance travelled, the number of stops and the amount of petrol put in at the end of the journey, and prints the average petrol consumption in kilometres per litre, rounded to the nearest litre.

```
      PROGRAM PETROL
      INTEGER STOPS, FILLUP
```

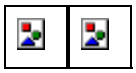
```
C
C THESE VARIABLES WOULD OTHERWISE BE TYPED REAL BY DEFAULT
C ANY TYPE SPECIFICATIONS MUST PRECEDE THE FIRST EXECUTABLE STATEMENT
C
        READ *, KM,STOPS,FILLUP
        USED = 40*STOPS + FILLUP
C COMPUTES THE PETROL USED AND CONVERTS IT TO REAL
        KPL = KM/USED + 0.5
C 0.5 IS ADDED TO ENSURE THAT THE RESULT IS ROUNDED
        PRINT *, 'AVERAGE KPL WAS',KPL

END
```

Figure 6: Petrol consumption program

This program illustrates some of the points about type conversion made in the previous chapter. In line 8, the number of litres of petrol used is computed. The computed value is of type INTEGER, but is converted to REAL when assigned to the REAL variable USED. In line 10, the expression KM/USED is evaluated as REAL, but would be truncated, not rounded, when assigned to the INTEGER variable KPL. Adding 0.5 before truncating has the effect of rounding up or down. This is a useful rounding method. It is illustrated further below.

KM/USED	KM/USED + 0.5	KPL
12.0	12.5	12
12.4	12.9	12
12.5	13.0	13
12.9	13.4	13



Copyright Notice and Credits

These notes were written by John Porter of the University of Strathclyde Computer Centre. They form the basis of the Computer Centre's Fortran 77 course. John can be reached at J.R.Porter@strath.ac.uk.

Contact the Computer Centre if you're interested in finding out about or attending any of our courses.

(webperson@strath.ac.uk)