# 16

# Control Structures

"Summarizing: as a slow-witted human being I have a very small head and I had better learn to live with it and to respect my limitations and give them full credit, rather than try to ignore them, for the latter vain effort will be punished by failure."

Edsger W. Dijkstra, *Structured Programming*

## Aims

The aims of this chapter are to introduce:

- Selection among various courses of action as part of the algorithm.
- The concepts and statements in Fortran needed to support the above:
  - Logical expressions and logical operators.
  - One or more *blocks* of statements.
- The IF THEN ENDIF construct.
- The IF THEN ELSE IF ENDIF construct.
- To introduce the CASE statement with examples.
- To introduce the DO loop, in three forms with examples, in particular:
  - The iterative DO loop.
  - The DO WHILE form.
  - The DO ... IF THEN EXIT END DO or repeat until form.
  - The CYCLE statement.
  - The EXIT statement.

# 16 Control Structures

When we look at this area it is useful to gain some historical perspective concerning the control structures that are available in a programming language.

At the time of the development of Fortran in the 1950s there was little theoretical work around and the control structures provided were very primitive and closely related to the capability of the hardware.

At the time of the first standard in 1966 there was still little published work regarding structured programming and control structures. The seminal work by Dahl, Dijkstra and Hoare was not published until 1972.

By the time of the second standard there was a major controversy regarding languages with poor control structures like Fortran which essentially were limited to the GOTO statement. The facilities in the language had led to the development and continued existence of major code suites that were unintelligible, and the pejorative term *spaghetti* was applied to these programs. Developing an understanding of what a program did became an almost impossible task in many cases.

Fortran missed out in 1977 on incorporating some of the more modern and intelligible control structures that had emerged as being of major use in making code easier to understand and modify.

It was not until the 1990 standard that a reasonable set of control structures had emerged and became an accepted part of the language. The more inquisitive reader is urged to read at least the work by Dahl, Dijkstra and Hoare to develop some understanding of the importance of control structures and the role of structured programming. The paper by Knuth is also highly recommended as it provides a very balanced coverage of the controversy of earlier times over the GOTO statement.

## 16.1 Selection among courses of action

In most problems you need to choose among various courses of action, e.g.,

- If overdrawn, then do not draw money out of the bank.
- If Monday, Tuesday, Wednesday, Thursday or Friday, then go to work.
- If Saturday, then go to watch Queens Park Rangers.
- If Sunday, then lie in bed for another two hours.

As most problems involve selection between two or more courses of action it is necessary to have the concepts to support this in a programming language. Fortran has a variety of selection mechanisms, some of which are introduced below.

### 16.1.1 The BLOCK IF statement

The following short example illustrates the main ideas:

```
. wake up
.
. check the date and time
IF (Today = = Sunday) THEN
        .
        . lie in bed for another two hours
        .
ENDIF
.
. get up
. make breakfast
```

If today is Sunday then the block of statements between the IF and the ENDIF is executed. After this block has been executed the program continues with the statements after the ENDIF. If today is not Sunday the program continues with the statements after the ENDIF immediately. This means that the statements after the ENDIF are executed whether or not the expression is true. The general form is:

```
IF (Logical expression) THEN
        .
        Block of statements
        .
ENDIF
```

The logical expression is an expression that will be either true or false; hence its name. Some examples of logical expressions are given below:

```
(Alpha >= 10.1)
```

Test if Alpha is greater than or equal to 10.1

```
(Balance <= 0.0)
```

Test if overdrawn

```
(( Today == Saturday).OR.( Today == Sunday))
```

Test if today is Saturday or Sunday

```
((Actual - Calculated) <= 1.0E-6)
```

Test if Actual minus Calculated is less than or equal to 1.0E-6

Fortran has the following relational and logical operators:

| Operator | Meaning | Type |
|---|---|---|
| = = | Equal | Relational |
| /= | Not equal | Relational |
| >= | Greater than or equal | Relational |
| <= | Less than or equal | Relational |
| < | Less than | Relational |
| > | Greater than | Relational |
| .AND. | And | Logical |
| .OR. | Or | Logical |
| .NOT. | Not | Logical |

The first six should be self-explanatory. They enable expressions or variables to be compared and tested. The last three enable the construction of quite complex comparisons, involving more than one test; in the example given earlier there was a test to see whether today was Saturday or Sunday.

Use of logical expressions and logical variables (something not mentioned so far) is covered again in a later chapter on logical data types.

The 'IF *expression* THEN *statements* ENDIF' is called a BLOCK IF construct. There is a simple extension to this provided by the ELSE statement. Consider the following example:

```
IF (Balance > 0.0) THEN
        . draw money out of the bank
ELSE
        . borrow money from a friend
ENDIF
Buy a round of drinks.
```

In this instance, one or other of the blocks will be executed. Then execution will continue with the statements after the ENDIF statement (in this case *buy a round*).

There is yet another extension to the BLOCK IF which allows an ELSEIF statement. Consider the following example:

```
IF (Today == Monday) THEN
   .
ELSEIF (Today == Tuesday) THEN
```

```
   .
ELSEIF (Today == Wednesday) THEN
   .
ELSEIF (Today == Thursday) THEN
   .
ELSEIF (Today == Friday) THEN
   .
ELSEIF (Today == Saturday) THEN
   .
ELSEIF (Today == Sunday) THEN
   .
ELSE
   there has been an error. The variable Today has
   taken on an illegal value.
ENDIF
```

Note that as soon as one of the logical expressions is true, the rest of the test is skipped, and execution continues with the statements after the ENDIF. This implies that a construction like

```
IF(I < 2)THEN
   ...
ELSEIF(I < 1)THEN
   ...
ELSE
   ...
ENDIF
```

is inappropriate. If I is less than 2, the latter condition will never be tested. The ELSE statement has been used here to aid in trapping errors or exceptions. This is recommended practice. A very common error in programming is to assume that the data are in certain well-specified ranges. The program then fails when the data go outside this range. It makes no sense to have a day other than Monday, Tuesday, Wednesday, Thursday, Friday, Saturday or Sunday.

### 16.1.2 Example 1: Quadratic roots

A quadratic equation is:

$$a\,x^2 + b\,x + c = 0$$

This program is straightforward, with a simple structure. The roots of the quadratic are either real, equal and real, or complex depending on the magnitude of the term B ** 2 – 4 * A * C. The program tests for this term being greater than or less than

zero: it assumes that the only other case is equality to zero (from the mechanics of a computer, floating point equality is rare, but we are safe in this instance):

```
PROGRAM ch1601
IMPLICIT NONE
REAL :: A , B , C , Term , A2 , Root1 , Root2
!
!    a b and c are the coefficients of the terms
!    a*x**2+b*x+c
!    find the roots of the quadratic, root1 and root2
!
   PRINT*,' GIVE THE COEFFICIENTS A, B AND C'
   READ*,A,B,C
   Term = B*B - 4.*A*C
   A2 = A*2.
! if term < 0, roots are complex
! if term = 0, roots are equal
! if term > 0, roots are real and different
   IF(Term < 0.0)THEN
      PRINT*,' ROOTS ARE COMPLEX'
   ELSEIF(Term > 0.0)THEN
      Term = SQRT(Term)
      Root1 = (-B+Term)/A2
      Root2 = (-B-Term)/A2
      PRINT*,' ROOTS ARE ',Root1,' AND ',Root2
   ELSE
      Root1 = -B/A2
      PRINT*,' ROOTS ARE EQUAL, AT ',Root1
   ENDIF
END PROGRAM ch1601
```

### 16.1.3 Note

Given the understanding you now have about real arithmetic and finite precision will the ELSE block above ever be executed?

### 16.1.4 Example 2: Date calculation

This next example is also straightforward. It demonstrates that, even if the conditions on the IF statement are involved, the overall structure is easy to determine. The comments and the names given to variables should make the program self-explanatory. Note the use of integer division to identify leap years:

```
PROGRAM ch1602
IMPLICIT NONE
```

```
INTEGER :: Year , N , Month , Day , T
!
! calculates day and month from year and
! day-within-year
! t is an offset to account for leap years.
! Note that the first criteria is division by 4
! but that centuries are only
! leap years if divisible by 400
! not 100 (4 * 25) alone.
!
   PRINT*,' year, followed by day within year'
   READ*,Year,N
!    checking for leap years
   IF ((Year/4)*4 == Year ) THEN
     T=1
     IF ((Year/400)*400 == Year ) THEN
       T=1
     ELSEIF ((Year/100)*100 == Year) THEN
       T=0
     ENDIF
   ELSE
     T=0
   ENDIF
!     accounting for February
   IF(N > (59+T))THEN
     Day=N+2-T
   ELSE
     Day=N
   ENDIF
   Month=(Day+91)*100/3055
   Day=(Day+91)-(Month*3055)/100
   Month=Month-2
   PRINT*,' CALENDAR DATE IS ', Day , Month , Year
END PROGRAM ch1602
```

### 16.1.5 The CASE statement

The CASE statement provides a very clear and expressive selection mechanism between two or more courses of action. Strictly speaking it could be constructed from the IF THE ELSE IF ENDIF statement, but with considerable loss of clarity. Remember that programs have to be read and understood by both humans and compilers!

### 16.1.6 Example 3: Simple calculator

```
PROGRAM ch1603
IMPLICIT NONE
!
! Simple case statement example
!

INTEGER :: I,J,K
CHARACTER :: Operator
  DO
    PRINT *,' Type in two integers'
    READ *, I,J
    PRINT *,' Type in operator'
    READ '(A)',Operator
    Calculator : &
    SELECT CASE (Operator)
      CASE ('+') Calculator
        K=I+J
        PRINT *,' Sum of numbers is ',K
      CASE ('-') Calculator
        K=I-J
        PRINT *,' Difference is ',K
      CASE ('/') Calculator
        K=I/J
        PRINT *,' Division is ',K
      CASE ('*') Calculator
      K=I*J
      PRINT *,' Multiplication is ',K
    CASE DEFAULT Calculator
      EXIT
    END SELECT Calculator
  END DO
END PROGRAM ch1603
```

The user is prompted to type in two integers and the operation that they would like carried out on those two integers. The CASE statement then ensures that the appropriate arithmetic operation is carried out. The program terminates when the user types in any character other than +, -, * or /.

The CASE DEFAULT option introduces the EXIT statement. This statement is used in conjunction with the DO statement. When this statement is executed control passes to the statement immediately after the matching END DO statement. In

the example above the program terminates, as there are no executable statements after the END DO.

### 16.1.7 Example 4: Counting vowels, consonants, etc.

This example is more complex, but again is quite easy to understand. The user types in a line of text and the program produces a summary of the frequency of the characters typed in:

```
PROGRAM ch1604
IMPLICIT NONE
!
! Simple counting of vowels, consonants,
! digits, blanks and the rest
!
INTEGER :: Vowels=0 , Consonants=0, Digits=0
INTEGER :: Blank=0, Other=0, I
CHARACTER :: Letter
CHARACTER (LEN=80) :: Line
  READ '(A)', Line
  DO I=1,80
    Letter=Line(I:I)
! the above extracts one character at position I
    SELECT CASE (Letter)
      CASE ('A','E','I','O','U', &
            'a','e','i','o','u')
        Vowels=Vowels + 1
      CASE ('B','C','D','F','G','H', &
            'J','K','L','M','N','P', &
            'Q','R','S','T','V','W', &
            'X','Y','Z',             &
            'b','c','d','f','g','h', &
            'j','k','l','m','n','p', &
            'q','r','s','t','v','w', &
            'x','y','z')
        Consonants=Consonants + 1
      CASE ('1','2','3','4','5','6','7','8','9','0')
        Digits=Digits + 1
      CASE (' ')
        Blank=Blank + 1
      CASE DEFAULT
        Other=Other+1
    END SELECT
  END DO
```

```
   PRINT *, ' Vowels =   ', Vowels
   PRINT *, ' Consonants =   ', Consonants
   PRINT *, ' Digits = ', Digits
   PRINT *, ' Blanks = ',Blank
   PRINT *, ' Other characters = ', Other
END PROGRAM ch1604
```

## 16.2 The three forms of the DO statement

You have already been introduced in the chapters on arrays to the iterative form of the DO loop, i.e.,

DO Variable = Start, End, Increment

block of statements

END DO

A complete coverage of this form is given in the three chapters on arrays.

There are two additional forms of the block DO that complete our requirements:

DO WHILE (Logical Expression)

block of statements

ENDDO

and

DO

block of statements

IF (Logical Expression) EXIT

END DO

The first form is often called a WHILE loop as the block of statements executes whilst the logical expression is true, and the second form is often called a REPEAT UNTIL loop as the block of statements executes until the statement is true.

Note that the WHILE block of statements may never be executed, and the REPEAT UNTIL block will always be executed at least once.

### 16.2.1 Example 5: Sentinel usage

The following example shows a complete program using this construct:

```
PROGRAM ch1605
IMPLICIT NONE
! this program picks up the first occurrence
```

```
! of a number in a list.
! a sentinel is used, and the array is 1 more
! than the max size of the list.
INTEGER , ALLOCATABLE , DIMENSION(:) :: A
INTEGER :: Mark
INTEGER :: I,Howmany
   OPEN (UNIT=1,FILE='DATA')
   PRINT *,' What number are you looking for?'
   READ  *, Mark
   PRINT *,' How many numbers to search?'
   READ  *,Howmany
   ALLOCATE(A(1:Howmany+1))
   READ(UNIT=1,FMT=*) (A(i),I=1,Howmany)
   I=1
   A(Howmany+1)= Mark
   DO WHILE(Mark /= A(I))
      I=I+1
   END DO
   IF(I == (Howmany+1)) THEN
      PRINT*,' ITEM NOT IN LIST'
   ELSE
      PRINT*,' ITEM IS AT POSITION ',I
   ENDIF
END PROGRAM ch1605
```

The *repeat until* construct is written in Fortran as:

```
   DO
   ...
   ...
      IF (Logical Expression) EXIT
   END DO
```

There are problems in most disciplines that require a numerical solution. The two main reasons for this are either that the problem can only be solved numerically or that an analytic solution involves too much work. Solutions to this type of problem often require the use of the *repeat until* construct. The problem will typically require the repetition of a calculation until the answers from successive evaluations differ by some small amount, decided generally by the nature of the problem. A program extract to illustrate this follows:

```
   REAL , PARAMETER :: TOL=1.0E-6
   .
```

```
DO
   ...
   CHANGE=
   ...
   IF (CHANGE <= TOL) EXIT
END DO
```

Here the value of the tolerance is set to 1.0E–6. Note again the use of the EXIT statement. The DO END DO block is terminated and control passes to the statement immediately after the matching END DO.

### 16.2.2 CYCLE and EXIT

These two statements are used in conjunction with the block DO statement. You have seen examples above of the use of the EXIT statement to terminate the block DO, and pass control to the statement immediately after the corresponding END DO statement.

The CYCLE statement can appear anywhere in a block DO and will immediately pass control to the start of the block DO. Examples of CYCLE and EXIT are given in later chapters.

### 16.2.3 Example 6: e**x evaluation

The function etox illustrates one use of the *repeat until* construct. The function evaluates e**x. This may be written as

$$1 + x/1! + x^2/2! + x^3/3! \ldots$$

or

$$1+\sum_{n=1}^{\infty} \frac{x^{n-1}}{(n-1)!}\frac{x}{n}$$

Every succeeding term is just the previous term multiplied by *x/n*. At some point the term *x/n* becomes very small, so that it is not sensibly different from zero, and successive terms add little to the value. The function therefore repeats the loop until *x/n* is smaller than the tolerance. The number of evaluations is not known beforehand, since this is dependent on *x*:

```
REAL FUNCTION etox(X)
IMPLICIT NONE
REAL :: Term
REAL , INTENT(IN) :: X
INTEGER :: Nterm
REAL , PARAMETER ::Tol = 1.0E-6
  etox=1.0
```

```
   Term=1.0
   Nterm=0
   DO
      Nterm = Nterm +1
      Term =( X / Nterm) * Term
      etox = etox + Term
      IF(ABS(Term) <= Tol)EXIT
   END DO
END FUNCTION etox

program ch1606
implicit none
real :: etox
real , parameter :: x=1.0
real :: y
   print *,' Fortran intrinsic ',exp(x)
   y=etox(x)
   print *,' User defined etox ',y
end program ch1606
```

The whole program compares the user defined function with the Fortran intrinsic exp function.

### 16.2.4 Example 7: Wave breaking on an offshore reef

This example is drawn from a situation where a wave breaks on an offshore reef or sand bar, and then reforms in the near-shore zone before breaking again on the coast. It is easier to observe the heights of the reformed waves reaching the coast than those incident to the terrace edge.

Both types of loops are combined in this example. The algorithm employed here finds the zero of a function. Essentially, it finds an interval in which the zero must lie; the evaluations on either side are of different signs. The *while loop* ensures that the evaluations are of different signs, by exploiting the knowledge that the incident wave height must be greater than the reformed wave height (to give the lower bound). The upper bound is found by experiment, making the interval bigger and bigger. Once the interval is found, its mean is used as a new potential bound. The zero must lie on one side or the other; in this fashion, the interval containing the zero becomes smaller and smaller, until it lies within some tolerance. This approach is rather plodding and unexciting, but is suitable for a wide range of problems

Here is the program:

```
PROGRAM Break
IMPLICIT NONE
REAL :: Hi , Hr , Hlow , High , Half , Xl
REAL :: Xh , Xm , D
REAL , PARAMETER :: Tol=1.0E-6
! problem - find hi from expression given
! in function f
! F=A*(1.0-0.8*EXP(-0.6*C/A))-B
! HI IS INCIDENT WAVE HEIGHT          (C)
! HR IS REFORMED WAVE HEIGHT          (B)
! D IS WATER DEPTH AT TERRACE EDGE    (A)
   PRINT*,' Give reformed wave height, and water depth'
   READ*,Hr,d
!
! for Hlow- let Hlow=hr
! for high- let high=Hlow*2.0
!
! check that signs of function results are different
!
   Hlow = Hr
   High = Hlow*2.0
   Xl = F( Hlow, Hr, D)
   Xh = F( High, Hr, D)
!
   DO WHILE ((XL*XH)  >= 0.0)
      HIGH = HIGH*2.0
      XH = F(HIGH,HR,D)
   END DO
!
   DO
      HALF=(HLOW+HIGH)*0.5
      XM=F(HALF,HR,D)
      IF((XL*XM) < 0.0)THEN
         XH=XM
         HIGH=HALF
      ELSE
         XL=XM
         HLOW=HALF
      ENDIF
      IF(ABS(HIGH-HLOW)<= TOL)EXIT
   END DO
   PRINT*,' Incident Wave Height Lies Between'
```

```
   PRINT*,Hlow,' and ',High,' metres'
CONTAINS
REAL FUNCTION F(A,B,C)
IMPLICIT NONE
REAL , INTENT (IN) :: A
REAL , INTENT (IN) :: B
REAL , INTENT (IN) :: C
   F=A*(1.0-0.8*EXP(-0.6*C/A))-B
END FUNCTION F
END PROGRAM Break
```

## 16.3 Summary

You have been introduced in this chapter to several control structures and these include:

- The *block if*.
- The *if then else if*.
- The *case* construct.
- The block *do* in three forms:
  - The *iterative do* or *do variable=start,end,increment ... end do*.
  - The *while* construct, or *do while ... end do*.
  - The *repeat until* construct, or *do ... if then exit end do*.
- The *cycle* and *exit* statements, which can be used with *do* statement in all three forms:
  - The *do variable = start,end,increment ... end do*.
  - The *while* construct, or *do while ... end do*.
  - The *repeat until* construct, or *do ... if then exit end do.*

These constructs are sufficient for solving a wide class of problems. There are other control statements available in Fortran, especially those inherited from Fortran 66 and Fortran 77, but those covered here are the ones preferred. We will look in Chapter 28 at one more control statement, the so-called GOTO statement, with recommendations as to where its use is appropriate.

### 16.3.1 Control structure formal syntax

**CASE**

```
SELECT CASE ( case variable )
   [ CASE case selector
      [executable construct ] ... ] ...
   [ CASE DEFAULT
   [executable construct ]
END SELECT
```

**DO**

```
DO [ label ]
   [executable construct ] ...
do termination

DO [ label ] [ , ] loop variable = initial value ,
final value , [ increment ]
   [executable construct ] ...
do termination

DO [ label ] [ , ] WHILE (scalar logical expression )
   [executable construct ] ...
do termination
```

**IF**

```
IF ( scalar logical expression ) THEN
   [executable construct ] ...
[ ELSE IF ( scalar logical expression  THEN
   [executable construct ] ... ] ...]
[ ELSE
      [executable construct ] ...]
END IF
```

## 16.4 Problems

1. Rewrite the program for the period of a pendulum. The new program should print out the length of the pendulum and period, for pendulum lengths from 0 to 100 cm in steps of 0.5 cm. The program should incorporate a function for the evaluation of the period.

2. Write a program to read an integer that must be positive.

**Hint**. Use a DO WHILE to make the user re-enter the value.

3. Using functions, do the following:

- Evaluate $n!$ from $n = 0$ to $n = 10$.

- Calculate 76!
- Now calculate (*x***n*)/*n*!, with $x = 13.2$ and $n = 20$.
- Now do it another way.

4. The program BREAK is taken from a real example. In the particular problem, the reformed wave height was 1 metre, and the water depth at the reef edge was 2 metres. What was the incident wave height? Rather than using an absolute value for the tolerance, it might be more realistic to use some value related to the reformed wave height. These heights are unlikely to be reported to better than about 5% accuracy. Wave energy may be taken as proportional to wave height squared for this example. What is the reduction in wave energy as a result of breaking on the reef or bar for this particular case.

5. What is the effect of using INT on negative real numbers? Write a program to demonstrate this.

6. How would you find the nearest integer to a real number? Now do it another way. Write a program to illustrate both methods. Make sure you test it for negative as well as positive values.

7. The function etox has been given in this chapter. The standard Fortran function EXP does the same job. Do they give the same answers? Curiously the Fortran standard does not specify how a *standard* function should be evaluated, or even how accurate it should be.

The physical world has many examples in which processes require that some threshold be overcome before they begin operation: critical mass in nuclear reactions, a given slope to be exceeded before friction is overcome, and so on. Unfortunately, most of these sorts of calculations become rather complex and not really appropriate here. The following problem tries to restrict the range of calculation, whilst illustrating the possibilities of decision making.

8. If a cubic equation is expressed as

$$z^3 + a_2 z^2 + a_1 z + a_0 = 0$$

and we let

$$q = \frac{a_1}{3} - \frac{(a_2 * a_2)}{9}$$

and

$$r = \frac{(a_1 * a_2 - 3 * a_0)}{6} - \frac{(a_2 * a_2 * a_2)}{27}$$

we can determine the nature of the roots as follows:

$q^3 + r^2 > 0$; one real root and a pair of complex
$q^3 + r^2 = 0$; all roots real, and at least two equal
$q^3 + r^2 < 0$; all roots real

Incorporate this into a suitable program, to determine the nature of the roots of a cubic from suitable input.

9. The form of breaking waves on beaches is a continuum, but for convenience we commonly recognise three major types: surging, plunging and spilling. These may be classified empirically by reference to the wave period, $T$ (seconds), the breaker wave height, $H_b$ (metres), and the beach slope, $m$. These three variables are combined into a single parameter, $B$, where

$$B = H_b/(gmT^2)$$

$g$ is the gravitational constant (981 cm $s^{-2}$). If $B$ is less than 0.003, the breakers are surging; if $B$ is greater than 0.068, they are spilling, and between these values, plunging breakers are observed.

(i) On the east coast of New Zealand, the normal pattern is swell waves, with wave heights of 1 to 2 metres and wave periods of 10 to 15 seconds. During storms, the wave period is generally shorter, say 6 to 8 seconds, and the wave heights higher, 3 to 5 metres. The beach slope may be taken as about 0.1. What changes occur in breaker characteristics as a storm builds up?

(ii) Similarly, many beaches have a concave profile. The lower beach generally has a very low slope, say less than 1 degree ($m$ = 0.018), but towards the high-tide mark, the slope increases dramatically, to say 10 degrees or more ($m$ = 0.18). What changes in wave type will be observed as the tide comes in?

## 16.5 Bibliography

Dahl O.J., Dijkstra E.W., Hoare C.A.R., *Structured Programming*, Academic Press, 1972.

- This is the original text, and a must. The quote at the start of the chapter by Dijkstra summarises beautifully our limitations when programming and the discipline we must have to master programming successfully.

Knuth D.E., *Structured Programming with GOTO Statements,* in Current Trends in Programming Methodology, Volume 1, Prentice-Hall, 1977.

- The chapter by Knuth provides a very succinct coverage of the arguments for the adoption of structured programming, and dispels many of the myths concerning the use of the GOTO statement. Highly recommended.