Chapter 13
Control Structures and Execution Control

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:
  - execution control.
  - executable constructs containing blocks.
    the associate construct.
    the block construct.
    the do construct.
    the if construct.
    the select case construct.
    the select rank construct.
    the select type construct.
  - Logical expressions and logical operators.
  - One or more blocks of statements.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_13

13.1 Introduction

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.

13.2 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.

13.3 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
```

Table 13.1 lists the Fortran logical and relational operators.

Table 13.1 Fortran logical and relational 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.

13.3.1 Example 1: Quadratic Roots

A quadratic equation is:

$$ax^2 + bx + c = 0$$

This program has 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 ch1301
  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'
  else if (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
  end if
end program ch1301
```

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

13.3.2 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 ch1302
  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
    else if ((year/100)*100==year) then
      t = 0
    end if
  else
    t = 0
  end if
! accounting for February
  if (n>(59+t)) then
    day = n + 2 - t
  else
    day = n
  end if
  month = (day+91)*100/3055
  day = (day+91) - (month*3055)/100
  month = month - 2
  print *, ' calendar date is ', day, month, &
    year
end program ch1302
```

13.4 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 then else if endif` statement, but with considerable loss of

clarity. Remember that programs have to be read and understood by both humans and compilers!

13.4.1 Example 3: Simple Calculator

```
program ch1303
  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 ch1303
```

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`.

13.4.2 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 ch1304
  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 ch1304
```

13.5 The Various 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 a number of additional forms of the block do that complete our requirements:

```
do while (logical expression)
  block of statements
enddo

do concurrent
  block of statements
enddo

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.

13.5.1 Example 5: Sentinel Usage

The following example shows a complete program using this construct:

```
program ch1305
  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.txt',status='old')
  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
  end if
end program ch1305
```

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`.

13.5.2 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 the next two examples, and later chapters in the book.

*13.5.3 Example 6: The Evaluation of e**x*

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

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

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:

```
module etox_module
  implicit none

contains
  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
end module etox_module

program ch1306
  use etox_module
  implicit none
  real, parameter :: x = 1.0
  real :: y

  print *, ' Fortran intrinsic ', exp(x)
  y = etox(x)
  print *, ' User defined etox ', y
end program ch1306
```

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

13.5.4 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 ch1307
  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
! The above is a Fortran 77
! statement function.
! 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
    end if
    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 ch1307
```

13.6 Do Concurrent

Here is some of the formal syntax of do loops taken from the standard.

```
loop-control is [ , ] do-variable =
```

```
    scalar-int-expr,
    scalar-int-expr
    [ , scalar-int-expr ]

  or [ , ] WHILE ( scalar-logical-expr )

  or [ , ] CONCURRENT
      concurrent-header
      concurrent-locality

  do-variable is scalar-int-variable-name

  The do-variable shall be a variable of type integer.

  concurrent-header is ( [ integer-type-spec :: ]
    concurrent-control-list
    [ , scalar-mask-expr ] )

  concurrent-control is index-name =
    concurrent-limit :
    concurrent-limit [ : concurrent-step ]

  concurrent-limit is scalar-int-expr
```

Here are the rules that apply to the `do concurrent` loop control.

- The concurrent-limit and concurrent-step expressions in the concurrent-control-list are evaluated. These expressions may be evaluated in any order. The set of values that a particular index-name variable assumes is determined as follows.
  - The lower bound m1, the upper bound m2, and the step m3 are of type integer with the same kind type parameter as the index-name. Their values are established by evaluating the first concurrent-limit, the second concurrent-limit, and the concurrent-step expressions, respectively, including, if necessary, conversion to the kind type parameter of the index-name according to the rules for numeric conversion (`Table 10.9` from the current standard). If concurrent-step does not appear, m3 has the value 1. The value m3 shall not be zero.
  - Let the value of max be (m2 m1 + m3)/m3. If max 0 for some index-name, the execution of the construct is complete. Otherwise, the set of values for the index-name is m1 + (k 1) m3 where k = 1, 2, …, max.
- The set of combinations of index-name values is the Cartesian product of the sets defined by each triplet specification. An index-name becomes defined when this set is evaluated.

- The scalar-mask-expr, if any, is evaluated for each combination of index-name values. If there is no scalar-mask-expr, it is as if it appeared with the value true. The index-name variables may be primaries in the scalar-mask-expr.
- The set of active combinations of index-name values is the subset of all possible combinations for which the scalar-mask-expr has the value true.

Note that the index-name variables can appear in the mask, for example

```
DO CONCURRENT (I=1:10, J=1:10, &
  A(I) > 0.0 .AND. B(J) < 1.0)
. . .
```

The following example illustrates a case in which the user knows that there are no repeated values in the index array IND. The DO CONCURRENT construct makes it easier for the processor to generate vector gather/scatter code, unroll the loop, or parallelize the code for this loop, potentially improving performance.

```
INTEGER :: A(N),IND(N)
DO CONCURRENT (I=1:M)
  A(IND(I)) = I
END DO
```

The following code demonstrates the use of the LOCAL clause so that the X inside the DO CONCURRENT construct is a temporary variable, and will not affect the X outside the construct.

```
X = 1.0
DO CONCURRENT (I=1:10) LOCAL (X)
  IF (A (I) > 0) THEN
    X = SQRT (A (I))
    A (I) = A (I) - X**2
  END IF
  B (I) = B (I) - A (I)
END DO
PRINT *, X ! Always prints 1.0.
```

A complete example of the `do concurrent` statement can be found in the chapter on OpenMP programming. FThe examples compares the performance of four ways of solving the same problem in Fortran using whole array syntax, a traditional simple do loop, a do concurrent solution and a solution base on OpenMP usage.

13.7 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 the `do` statement
- The `do concurrent` statement.

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 Chap. 35 at one more control statement, the so-called goto statement, with recommendations as to where its use is appropriate.

13.7.1 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
```

13.8 Problems

13.1 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.

13.2 Write a program to read an integer that must be positive.
Hint. use a `do while` to make the user re-enter the value.

13.3 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.

13.4 The program ch1307 is taken from a real example. In the particular problem, the reformed wave height was 1 m, and the water depth at the reef edge was 2 m. 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.

13.5 What is the effect of using `int` on negative real numbers? Write a program to demonstrate this.

13.6 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.

13.7 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.

13.8 If a cubic equation is expressed as

$$ax^3 + bx^2 + cx + d = 0$$

and we let

$$\Delta = 18abcd - 4b^3d + b^2c^2 - 4ac^3 - 27a^2d^2$$

We can determine the nature of the roots as follows
$\Delta > 0$: three distinct real roots
$\Delta = 0$: has a multiple root and all roots are real
$\Delta < 0$: 1 real root and 2 non real complex conjugate roots
Incorporate this into a program, to determine the nature of the roots of a cubic from suitable input.

13.9 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 ($981\,\mathrm{cm\,s^{-2}}$). 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 m and wave periods of 10 to 15 s. During storms, the wave period is generally shorter, say 6 to 8 s, and the wave heights higher, 3 to 5 m. 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° (m = 0.018), but towards the high-tide mark, the slope increases dramatically, to say 10° or more (m = 0.18). What changes in wave type will be observed as the tide comes in?

13.9 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.

ISO/IEC DIS 1539-1 Information technology – Programming languages – Fortran – Part 1: Base language

- Fortran 2018 draft standard.

```
https://www.iso.org/standard/72320.html
```