

1. Repetition (DO loops)
  2. Conditional processing (IF, CASE)
  3. Arrays
  4. Functions and loops in different programming languages
- 

## 1. Repetition (DO Loops)

### Example 1.1 Different types of DO-Loop

Newton's method for square roots:

$$x_{n+1} = \frac{1}{2} \left( x_n + \frac{A}{x_n} \right) \quad \rightarrow \quad \sqrt{A}$$

(i) Deterministic DO loop – repeat a fixed number of times

```
PROGRAM NEWTON
  IMPLICIT NONE
  REAL A           ! number to be square-rooted
  REAL X           ! current value of root
  INTEGER N        ! loop counter

  PRINT *, 'Enter a number'
  READ *, A        ! input number to be rooted

  X = 1.0          ! initial value

  DO N = 1, 10     ! fixed number of iterations
    X = 0.5 * ( X + A / X ) ! update value
    PRINT *, X
  END DO

END PROGRAM NEWTON
```

(ii) Non-deterministic DO loops – repeat until some criterion is met.

(a) Using IF ( . . . ) EXIT

```
PROGRAM NEWTON
  IMPLICIT NONE
  REAL A                ! number to be square-rooted
  REAL X, XOLD          ! current and previous value
  REAL CHANGE           ! change during one iteration
  REAL, PARAMETER :: TOLERANCE = 1.0E-6 ! tolerance for convergence

  PRINT *, 'Enter a number'
  READ *, A             ! input number to be rooted

  X = 1.0               ! initial value

  DO
    XOLD = X            ! store previous value
    X = 0.5 * ( X + A / X ) ! update value
    PRINT *, X

    CHANGE = ABS( ( X - XOLD ) / X ) ! fractional change
    IF ( CHANGE < TOLERANCE ) EXIT ! criterion for stopping
  END DO

END PROGRAM NEWTON
```

(b) Using DO WHILE ( . . . )

```
PROGRAM NEWTON
  IMPLICIT NONE
  REAL A                ! number to be square-rooted
  REAL X, XOLD          ! current and previous value
  REAL CHANGE           ! change during one iteration
  REAL, PARAMETER :: TOLERANCE = 1.0E-6 ! tolerance for convergence

  PRINT *, 'Enter a number'
  READ *, A             ! input number to be rooted

  X = 1.0               ! initial value
  CHANGE = 1.0          ! anything big enough to make
                       ! the first loop run

  DO WHILE ( CHANGE > TOLERANCE ) ! criterion for continuing
    XOLD = X            ! store previous value
    X = 0.5 * ( X + A / X ) ! update value
    PRINT *, X

    CHANGE = ABS( ( X - XOLD ) / X ) ! fractional change
  END DO

END PROGRAM NEWTON
```

### Example 1.2 Summing power series.

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Note that each term is *not* worked out from scratch, but – more efficiently – as a multiple of the previous one:

$$\frac{x^n}{n!} = \frac{x}{n} \times \frac{x^{n-1}}{(n-1)!}$$

```
PROGRAM POWER_SERIES
  IMPLICIT NONE
  REAL, EXTERNAL :: NEW_EXP           ! declare a function to be used
  REAL VALUE                          ! number to test

  PRINT *, 'Enter a number'
  READ *, VALUE

  PRINT *, 'Sum of series = ', NEW_EXP( VALUE ) ! our own function
  PRINT *, 'Actual EXP(X) = ', EXP( VALUE )    ! standard function

END PROGRAM POWER_SERIES

!=====

REAL FUNCTION NEW_EXP( X )
! Sum a power series for exp(X)
  IMPLICIT NONE
  REAL X                               ! argument of function
  INTEGER N                             ! number of a term
  REAL TERM                             ! a term in the series
  REAL, PARAMETER :: TOLERANCE = 1.0E-07 ! truncation level

  ! First term
  N = 0; TERM = 1; NEW_EXP = TERM

  ! Add successive terms until they become negligible
  DO WHILE ( ABS( TERM ) > TOLERANCE ) ! criterion for continuing
    N = N + 1                          ! index of next term
    TERM = TERM * X / N                 ! new term is a multiple of last
    NEW_EXP = NEW_EXP + TERM           ! add to sum
  END DO

END FUNCTION NEW_EXP
```

**Warning:** the termination criterion used here:

$$|term| < \textit{small number}$$

is OK here because this power series can be shown to converge. This is *not* always a sufficient condition for convergence; for example, the harmonic series

$$\sum \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots$$

actually diverges, even though the terms tend to zero.

## 2. Conditional Processing (IF, CASE)

### Example 2.1 Comparing IF and CASE.

```
PROGRAM EXAM
  IMPLICIT NONE
  INTEGER MARK
  CHARACTER GRADE

  DO
    WRITE( *, '( "Enter mark (negative to end): " )', ADVANCE = 'NO' )
    READ *, MARK
    IF ( MARK < 0 ) STOP           ! stop program with a negative value

    IF ( MARK >= 70 ) THEN
      GRADE = 'A'
    ELSE IF ( MARK >= 60 ) THEN
      GRADE = 'B'
    ELSE IF ( MARK >= 50 ) THEN
      GRADE = 'C'
    ELSE IF ( MARK >= 40 ) THEN
      GRADE = 'D'
    ELSE IF ( MARK >= 30 ) THEN
      GRADE = 'E'
    ELSE
      GRADE = 'F'
    END IF
    PRINT *, 'Grade is ', GRADE
  END DO

END PROGRAM EXAM
```

```
PROGRAM EXAM
  IMPLICIT NONE
  INTEGER MARK
  CHARACTER GRADE

  DO
    WRITE( *, '( "Enter mark (negative to end): " )', ADVANCE = 'NO' )
    READ *, MARK
    IF ( MARK < 0 ) STOP           ! stop program with a negative value

    SELECT CASE ( MARK )
      CASE ( 70: )
        GRADE = 'A'
      CASE ( 60:69 )
        GRADE = 'B'
      CASE ( 50:59 )
        GRADE = 'C'
      CASE ( 40:49 )
        GRADE = 'D'
      CASE ( 30:39 )
        GRADE = 'E'
      CASE ( :29 )
        GRADE = 'F'
    END SELECT
    PRINT *, 'Grade is ', GRADE
  END DO

END PROGRAM EXAM
```

### 3. Arrays

#### Example 3.1 Illustrating operations element-by-element with arrays.

```
PROGRAM MATRIX
  IMPLICIT NONE
  REAL, DIMENSION(3,3) :: A, B, C      ! declare size of A, B and C
! REAL A(3,3), B(3,3), C(3,3)         ! alternative dimension statement
  REAL PI                               ! the number pi
  INTEGER I, J                           ! counters
  CHARACTER (LEN=*), PARAMETER :: FMT = '( A, 3(/, 3(1X, F8.3)), / )'
                                         ! format string for output

! Basic initialisation of matrices by assigning all values - inefficient
A(1,1) = 1.0;  A(1,2) = 2.0;  A(1,3) = 3.0
A(2,1) = 4.0;  A(2,2) = 5.0;  A(2,3) = 6.0
A(3,1) = 7.0;  A(3,2) = 8.0;  A(3,3) = 9.0

B(1,1) = 10.0; B(1,2) = 20.0; B(1,3) = 30.0
B(2,1) = 40.0; B(2,2) = 50.0; B(2,3) = 60.0
B(3,1) = 70.0; B(3,2) = 80.0; B(3,3) = 90.0

! Alternative initialisation using DATA statements - note order
! DATA A / 1.0, 4.0, 7.0, 2.0, 5.0, 8.0, 3.0, 6.0, 9.0 /
! DATA B / 10.0, 40.0, 70.0, 20.0, 50.0, 80.0, 30.0, 60.0, 90.0 /

! Alternative initialisation computing each element of A
! DO J = 1, 3
!   DO I = 1, 3
!     A(I,J) = (I - 1) * 3 + J
!   END DO
! END DO
! then whole-array operation for B
! B = 10.0 * A

! Write out matrices (using implied DO loops)
WRITE( *, FMT ) 'A', ( ( A(I,J), J = 1, 3 ), I = 1, 3 )
WRITE( *, FMT ) 'B', ( ( B(I,J), J = 1, 3 ), I = 1, 3 )

! Matrix sum
C = A + B
WRITE( *, FMT ) 'A+B', ( ( C(I,J), J = 1, 3 ), I = 1, 3 )

! "Element-by-element" multiplication
C = A * B
WRITE( *, FMT ) 'A*B', ( ( C(I,J), J = 1, 3 ), I = 1, 3 )

! "Proper" matrix multiplication
C = MATMUL( A, B )
WRITE( *, FMT ) 'MATMUL(A,B)', ( ( C(I,J), J = 1, 3 ), I = 1, 3 )

! Some operation applied to all elements of a matrix
PI = 4.0 * ATAN( 1.0 )
C = SIN( B * PI / 180.0 )
WRITE( *, FMT ) 'SIN(B)', ( ( C(I,J), J = 1, 3 ), I = 1, 3 )

END PROGRAM MATRIX
```

## Functions and Loops in Different Programming Languages

Consider a function

$$\text{sumsqr}(n) = 1^2 + 2^2 + \dots + n^2$$

(This could be worked out analytically, but the intention is to illustrate loops.)

### Fortran

```
Integer Function sumsqr( n )
  Integer n                ! declare argument type
  Integer i                ! declare internal variables

  sumsqr = 0              ! initialise sum

  Do i = 1, n              ! start of loop
    sumsqr = sumsqr + i * i ! add to sum
  End Do                  ! end of loop

End Function sumsqr
```

### Visual Basic

```
Function sumsqr(n As Integer) As Integer
  Dim i As Integer        ' declare internal variables

  sumsqr = 0              ' initialise sum

  For i = 1 To n          ' start of loop
    sumsqr = sumsqr + i * i ' add to sum
  Next i                  ' end of loop

End Function
```

### C++

```
int sumsqr( int n ) {
  int i, value;          // declare internal variables

  value = 0;             // initialise sum

  for (i = 1; i <= n; i++) { // start of loop
    value += i * i;       // add to sum
  }                       // end of loop

  return value;
}
```

Actually, the last is a little bit naughty (but typical of C++ shorthand):

`i++` is equivalent to `i = i + 1`  
`value += i * i` is equivalent to `value = value + i * i`