What is an Array?

An array is a collection of data of the <u>same</u> type. Array elements are <u>indexed</u> or <u>subscripted</u> like x_1, x_2, $\ldots$, x_n in mathematics.

Components of an Array

1. A **name**: `InputData`, `Sums`, etc.
2. A **type**: `INTEGER`, `REAL`, `LOGICAL` and `CHARACTER`.
3. An **extent**: `-10:10`, `0:100`, `50:100`.

   Extents have been used in `CASE` labels. The syntax of an extent is

   ```
   Lower-bound : Upper-bound
   ```

   where `Lower-bound` and `Upper-bound` are integers and `Lower-bound` must be less than or equal to `Upper-bound`.

Declaring an Array?

Syntax

```
type, DIMENSION ( extent ) :: name-1, name-2, ..., name-n
```

1. Two `REAL` arrays and one `INTEGER` array:

```
REAL, DIMENSION(-1:1)     :: a, Sum
INTEGER, DIMENSION(0:100) :: InputData
```

2. The lower bound and upper bound can be `PARAMETER`s:

```
INTEGER, PARAMETER :: MaximumSize = 100
LOGICAL, DIMENSION(1:MaximumSize) :: AnswerSheet

INTEGER, PARAMETER :: LowerBound = -10
INTEGER, PARAMETER :: UpperBound =  10
REAL, DIMENSION(LowerBound:UpperBound) :: Score, Mark
```

Array Elements

The syntax of an array element is

```
array-name ( integer-expression )
```

The `integer-expression` above must have a value in the range of the extent used to declare the array.

1. `AnswerSheet(3)`, `AnswerSheet(i+j)`.
2. `Score(-5)`, `Score(2*i-k)`, `Mark(10)`

Implied DO

Syntax

```
( item-1, item-2, ..., item-n, DO-var = initial, final, step )
( item-1, item-2, ..., item-n, DO-var = initial, final )
```

Semantics

For each possible value of the DO variable, all items (*i.e.*, `item-1`, `item-2`, ..., `item-n`) are listed once and adjacent items are separated by a comma.

1. The following produces -1, 0, 1, 2.

   ```
   ( i, i = -1, 2 )
   ```

2. The following produces 1, 1, 4, 16, 7, 49, 10, 100.

   ```
   ( i, i*i, i = 1, 10, 3 )
   ```

3. The following produces `a(1)`, `b(2)`, `a(2)`, `b(3)`, `a(3)`, `b(4)`.

   ```
   ( a(i), b(i+1), i = 1, 3 )
   ```

4. The following produces `3*a(3)`, `b(3)-c(2)`, 3, `0*a(0)`, `b(0)-c(-1)`, 0, `(-3)*a(-3)`, `b(-3)-c(-4)`, -3.

   ```
   ( k*a(k), b(k)-c(k-1), k, k = 3, -3, -3 )
   ```

Nested Implied DO

An implied `DO` can be an item of another implied `DO`. For a nested implied `DO`, it would be better to work from outside-in and treat an inner implied `DO` as an item.

1. Consider the following implied `DO`:

   ```
   ( i, ( i*j, j = 1, 3), i = 1, 3 )
   ```

   It is first expanded to the following by treating the inner implied `DO` as an item:

   ```
   1,  (1*j, j=1,3),  2,  (2*j, j=1,3),  3,  (3*j, j=1,3)
   ```

   Then, expanding the above yields:

   ```
   1, 1*1, 1*2, 1*3, 2, 2*1, 2*2, 2*3, 3, 3*1, 3*2, 3*3
   ```

2. Consider the following implied `DO`:

   ```
   ((a(i)*b(j), j=1, 2), i=1, 3)
   ```

   It is first expanded to the following by treating the inner implied `DO` as an item:

   ```
   (a(1)*b(j),j=1,2),  (a(2)*b(j),j=1,2),  (a(3)*b(j),j=1,2)
   ```

   Then, expanding the above yields:

   ```
   a(1)*b(1), a(1)*b(2), a(2)*b(1), a(2)*b(2),
   a(3)*b(1), a(3)*b(2)
   ```

Array Input/Output

Implied `DO` can be used in `READ(*,*)` and `WRITE(*,*)` statements. The generated items replace the implied `DO`.

1. The following two are equivalent:

```
READ(*,*)  (Value(I), I = 1, 5)
READ(*,*)  (x(I), Sum(I), I = 1, 3)

READ(*,*)  Value(1), Value(2), Value(3), Value(4), Value(5)
READ(*,*)  x(1), Sum(1), x(2), Sum(2), x(3), Sum(3)
```

2. **What is the difference?**
   Consider the following two `READ(*,*)`s:

```
DO i = 1, 5
   READ(*,*)  x(i)          READ(*,*)  (x(i), i=1, 5)
END DO
```

   If the input is

```
1
2
3
4
5
```

   Both work the same way.

```
DO i = 1, 5
   READ(*,*)  x(i)        READ(*,*)  (x(i), i=1, 5)
END DO
```

If the input is changed to

```
1  2  3  4  5
```

The left one will not work properly.

Reason: Each `READ(*,*)` consumes at least one line of input!

Short Examples

1. Clear an array to zero

```
INTEGER, PARAMETER :: LOWER = -100, UPPER = 100
INTEGER, DIMENSION(LOWER:UPPER) :: a
INTEGER                :: i

DO i = LOWER, UPPER
   a(i) = 0
END DO
```

2. Set an array element to its index or subscript

```
INTEGER, PARAMETER :: BOUND = 20
INTEGER, DIMENSION(1:BOUND) :: Array
INTEGER                :: i

DO i = 1, BOUND
   Array(i) = i
END DO
```

3. Odd indexed elements receive 1 and others receive zero

```
INTEGER, PARAMETER :: ARRAY_SIZE = 50
INTEGER, DIMENSION(1:ARRAY_SIZE) :: OddEven
INTEGER                :: Element

DO Element = 1, ARRAY_SIZE
   OddEven(Element) = MOD(Element, 2)
END DO
```

4. **Compute the sum of array element m to element n, where $m \leq n$ are input integers**

```
REAL, PARAMETER :: MAX_SIZE = 100
REAL, DIMENSION(-MAX_SIZE:MAX_SIZE) :: DataArray
REAL            :: Sum
INTEGER         :: m, n, k

READ(*,*)  m, n
Sum = 0.0
DO k = m, n
   Sum = Sum + DataArray(k)
END DO
```

5. **Compute the sum of the corresponding elements of two arrays into a third one**

```
REAL, PARAMETER :: LENGTH = 35
REAL, DIMENSION(1:LENGTH) :: A, B, C
INTEGER         :: Index

DO Index = 1, LENGTH
   C(Index) = A(Index) + B(Index)
END DO
```

6. **Find the larger element of the corresponding elements of two arrays and store it into a third one**

```
REAL, PARAMETER :: LENGTH = 35
REAL, DIMENSION(1:LENGTH) :: A, B, C
INTEGER         :: Index

DO Index = 1, LENGTH
   IF (A(Index) > B(Index) THEN
      C(Index) = A(Index)
   ELSE
      C(Index) = B(Index)
   END IF
END DO
```

7. **Compute the *inner product* of two arrays. The inner product of two arrays is the sum of all products of corresponding elements**

```
REAL, PARAMETER :: VECTOR_SIZE = 10
REAL, DIMENSION(1:VECTOR_SIZE) :: Vector1
REAL, DIMENSION(1:VECTOR_SIZE) :: Vector2
REAL            :: InnerProduct
INTEGER         :: Elements_Used, n

READ(*,*)  Elements_Used

InnerProduct = 0.0
DO n = 1, Elements_Used
   InnerProduct = InnerProduct + Vector1(n)*Vector2(n)
END DO
```

8. Find the smallest element and its location of an array

```
INTEGER, PARAMETER :: BEGIN = -100, END = 50
INTEGER, DIMENSION(BEGIN:END) :: Data
INTEGER             :: Minimum, Location
INTEGER             :: k

Minimum  = Data(BEGIN)
Location = BEGIN
DO k = BEGIN+1, END
   IF (Data(k) < Minimum) THEN
      Minimum  = Data(k)
      Location = k
   END IF
END DO

WRITE(*,*)  "The minimum is in position ", Location
WRITE(*,*)  "Minimum value is ", Minimum
```

9. **A simple modification to the previous example can find the minimum of a section of an array**

```
INTEGER, PARAMETER :: BEGIN = -100, END = 50
INTEGER, DIMENSION(BEGIN:END) :: Data
INTEGER              :: Left, Right
INTEGER              :: Minimum, Location
INTEGER              :: k

READ(*,*)  Left, Right
Minimum  = Data(Left)     ! **** changed ****
Location = Left
DO k = Left+1, Right
   IF (Data(k) < Minimum) THEN
      Minimum  = Data(k)
      Location = k
   END IF
END DO

WRITE(*,*)  "The minimum between ", Left, " and " &
            Right, " is in position ", Location
WRITE(*,*)  "Minimum value is ", Minimum
```

10. **The following code simulates marking a scanned answer sheet**

```
INTEGER, PARAMETER :: NO_OF_PROBLEMS = 20
INTEGER, DIMENSION(1:NO_OF_PROBLEMS) :: Solution
INTEGER, DIMENSION(1:NO_OF_PROBLEMS) :: Answer
INTEGER            :: i, Count, IO

READ(*,*)  (Solution(i), i=1, NO_OF_PROBLEMS)
DO
   READ(*,*,IOSTAT=IO)  (Answer(i), i=1, NO_OF_PROBLEMS)
   IF (IO < 0)  EXIT
   Count = 0
   DO i = 1, NO_OF_PROBLEMS
      IF (Solution(i) == Answer(i)) THEN
         Count = Count + 1
      END IF
   END DO
   WRITE(*,*)  (Answer(i), i=1, NO_OF_PROBLEMS), &
               REAL(Count)/NO_OF_PROBLEMS * 100.0
END DO
```

Computing Mean, Variance and Standard Deviation

$$\begin{aligned}
\text{Mean} &= \frac{1}{n}\sum_{i=1}^{n} x_i \\
\text{Variance} &= \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \text{Mean})^2 \\
\text{Standard Deviation} &= \sqrt{\text{Variance}}
\end{aligned}$$

```
PROGRAM  MeanVariance
   IMPLICIT  NONE
   INTEGER, PARAMETER :: MAX_SIZE = 50
   REAL, DIMENSION(1:MAX_SIZE) :: Data
   REAL                :: Mean, Variance, StdDev
   INTEGER             :: n, i

   READ(*,*)  n
   READ(*,*)  (Data(i), i = 1, n)
   WRITE(*,*)  "Input Data:"
   WRITE(*,*)  (Data(i), i = 1, n)

   Mean = 0.0
   DO i = 1, n
      Mean = Mean + Data(i)
   END DO
   Mean = Mean / n

   Variance = 0.0
   DO i = 1, n
      Variance = Variance + (Data(i) - Mean)**2
   END DO
   Variance = Variance / (n - 1)
   StdDev   = SQRT(Variance)
```

```
   WRITE(*,*)
   WRITE(*,*)  "Mean               : ", Mean
   WRITE(*,*)  "Variance           : ", Variance
   WRITE(*,*)  "Standard Deviation : ", StdDev
   WRITE(*,*)
   WRITE(*,*)  "Analysis Table:"
   DO i = 1, n
      IF (Data(i) > Mean + StdDev) THEN
         WRITE(*,*)  Data(i), Data(i) - Mean, "<-- Good"
      ELSE IF (Data(i) < Mean - StdDev) THEN
         WRITE(*,*)  Data(i), Data(i) - Mean, "<-- Bad"
      ELSE
         WRITE(*,*)  Data(i), Data(i) - Mean
      END IF
   END DO
END PROGRAM  MeanVariance
```

Computing Moving Average

Given x_1, x_2, ..., x_n and an integer $k > 0$, computing the moving averages y_1, y_2, ..., y_{n-k+1}

$$
\begin{aligned}
y_1 &= \frac{1}{k}(x_1 + x_2 + \cdots + x_k) \\
y_2 &= \frac{1}{k}(x_2 + x_3 + \cdots + x_{k+1} \\
y_3 &= \frac{1}{k}(x_3 + x_4 + \cdots + x_{k+2}) \\
&\vdots \\
y_{n-k+1} &= \frac{1}{k}(x_{n-k+1} + x_{n-k+2} + \cdots + x_n)
\end{aligned}
$$

```
PROGRAM  MovingAverage
   IMPLICIT  NONE
   INTEGER, PARAMETER :: MAX_SIZE = 30
   REAL, DIMENSION(1:MAX_SIZE) :: x, Avg
   REAL                   :: Sum
   INTEGER                :: Window, Size, i, j
   READ(*,*)  Size, (x(i), i = 1, Size)
   READ(*,*)  Window
   DO i = 1, Size-Window+1
      Sum = 0.0
      DO j = i, i+Window-1
         Sum = Sum + x(j)
      END DO
      Avg(i) = Sum / Window
   END DO
   WRITE(*,*)  "Moving Average of the Given Array:"
   WRITE(*,*)  (Avg(i), i = 1, Size-Window+1)
END PROGRAM  MovingAverage
```

Reversing an Array

Given an array, write a program to "reverse" its elements. That is, if the array contains 1, 3, 5, 7, 9, the new array is 9, 7, 5, 3, 1.

```
PROGRAM  Reverse
   IMPLICIT  NONE

   INTEGER, PARAMETER :: SIZE = 30
   INTEGER, DIMENSION(1:SIZE) :: a
   INTEGER            :: n, Head, Tail, Temp, i

   READ(*,*)  n
   READ(*,*)  (a(i), i = 1, n)
   WRITE(*,*) "Input array:"
   WRITE(*,*) (a(i), i = 1, n)

   Head = 1
   Tail = n
   DO
      IF (Head >= Tail)  EXIT
      Temp    = a(Head)
      a(Head) = a(Tail)
      a(Tail) = Temp
      Head    = Head + 1
      Tail    = Tail - 1
   END DO

   WRITE(*,*)  "Reversed array:"
   WRITE(*,*)  (a(i), i = 1, n)

END PROGRAM  Reverse
```

Palindrome?

Check if a array reads the same in both directions.

```
PROGRAM  Palindrome
   IMPLICIT  NONE

   INTEGER, PARAMETER :: LENGTH = 30
   INTEGER, DIMENSION(1:LENGTH)  :: x
   INTEGER              :: Size, Head, Tail, i

   READ(*,*) Size, (x(i), i = 1, Size)
   WRITE(*,*)  "Input array:"
   WRITE(*,*)  (x(i), i = 1, Size)

   Head = 1
   Tail = Size
   DO
      IF (Head >= Tail)  EXIT
      IF (x(Head) /= x(Tail))  EXIT
      Head = Head + 1
      Tail = Tail - 1
   END DO

   WRITE(*,*)
   IF (Head >= Tail) THEN
      WRITE(*,*) "The input array is a palindrome"
   ELSE
      WRITE(*,*) "The input array is NOT a palindrome"
   END IF

END PROGRAM  Palindrome
```

Sending Arrays to Functions and Subroutines – I

The whole extent, including lower bound and upper bound, are send to functions and subroutines with formal arguments.

Example 1

```
PROGRAM  Example
   IMPLICIT  NONE
   INTEGER, PARAMETER :: LOWER_BOUND = 20
   INTEGER, PARAMETER :: UPPER_BOUND = 50
   INTEGER, DIMENSION(LOWER_BOUND:UPPER_BOUND) :: Data
   REAL, DIMENSION(1:LOWER_BOUND)              :: Values
   LOGICAL, DIMENSION(21:UPPER_BOUND)           :: Answers
      ..........
   CALL  First(Data, Value, Answers, LOWER_BOUND, UPPER_BOUND, 21)
      ..........
CONTAINS
   SUBROUTINE  First(x, y, z, Lower, Upper, LL)
      IMPLICIT  NONE
      INTEGER, INTENT(IN)                           :: Lower
      INTEGER, INTENT(IN)                           :: Upper
      INTEGER, INTENT(IN)                           :: LL
      INTEGER, DIMENSION(Lower:Upper), INTENT(IN) :: x
      REAL, DIMENSION(1:Lower), INTENT(OUT)       :: y
      LOGICAL, DIMENSION(LL:Upper), INTENT(INOUT) :: z
            ..........
   END SUBROUTINE  First
END PROGRAM   Example
```

In many cases, not all elements of an array are used.

Example 2

```
PROGRAM  Test
   IMPLICIT  NONE
   INTEGER, PARAMETER :: MAX_SIZE = 1000
   REAL, DIMENSION(1:MAX_SIZE) :: Data
   INTEGER                     :: ActualSize
   INTEGER                     :: i

   READ(*,*) ActualSize
   READ(*,*) (Data(i), i=1, ActualSize)
   WRITE(*,*) "Sum = ", Sum(Data, ActualSize, MAX_SIZE)

CONTAINS
   REAL FUNCTION  Sum(x, n, SIZE)
      IMPLICIT  NONE
      REAL, INTENT(IN)                     :: SIZE, n
      REAL, DIMENSION(1:SIZE), INTENT(IN) :: x
      REAL                                 :: Total
      INTEGER                              :: i

      Total = 0.0
      DO i = 1, n
         Total = Total + x(i)
      END DO
      Sum = Total
   END FUNCTION  Sum
END PROGRAM  Test
```

Array elements are variables.

Example 3

```
PROGRAM  Elements
   IMPLICIT  NONE
   INTEGER, PARAMETER         :: BOUND_1 = 100
   INTEGER, PARAMETER         :: BOUND_2 = BOUND_1 - 2
   REAL, DIMENSION(1:BOUND_1) :: Input
   REAL, DIMENSION(1:BOUND_2) :: Avg
   INTEGER                    :: n, i

   READ(*,*) n, (Input(i), i=1, n)
   DO i = 1, n-2
      Avg(i) = Average(Input(i), Input(i+1), Input(i+2))
   END
   WRITE(*,*)  (Avg(i), i=1, n-2)

CONTAINS
   REAL FUNCTION  Average(a, b, c)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: a, b, c
      Average = (a + b + c)/3.0
   END FUNCTION  Average
END PROGRAM Elements
```

Example 4: Do not change these bounds

```
SUBROUTINE  Bad(x, m, n)
   IMPLICIT  NONE
   INTEGER, INTENT(INOUT)                 :: m, n
   INTEGER, DIMENSION(m:n), INTENT(INOUT) :: x
        ..........
   m = .....        ! BAD MOVE
   n = .....        ! BAD MOVE
END SUBROUTINE  Bad
```

Sending Arrays to Functions and Subroutines – II
Assumed-Shape Arrays

Syntax

Assume-shape arrays have an extent as follows:

```
lower-bound :
            :
```

It does not have an upper bound. If the lower bound is 1, it can be eliminated reducing to the second form.
If the actual argument is

```
INTEGER, DIMENSION(-2:2) :: x
```

then, array `x()` has the following 5 elements:

```
x(-2)   x(-1)   x(0)   x(1)   x(2)
```

If the formal argument is

```
INTEGER, DIMENSION(1:) :: y
```

then, array `y()` has the following 5 elements:

```
y(1)   y(2)   y(3)   y(4)   y(5)
```

Since the **size** of `y()` and `x()` are identical, `y()` has five elements. When `y(1)`, `y(2)`, ... are used, it is actually using `x(-2)`, `-1`, ...

Sending Arrays to Functions and Subroutines – II
Assumed-Shape Arrays

Let us rewrite the previous examples. Since it is not necessary to pass upper bounds, the argument lists become shorter.

Example 1

```
PROGRAM  Example
   IMPLICIT  NONE
   INTEGER, PARAMETER :: LOWER_BOUND = 20
   INTEGER, PARAMETER :: UPPER_BOUND = 50
   INTEGER, DIMENSION(LOWER_BOUND:UPPER_BOUND) :: Data
   REAL, DIMENSION(1:LOWER_BOUND)              :: Values
   LOGICAL, DIMENSION(21:UPPER_BOUND)          :: Answers
      ..........
   CALL  First(Data, Value, Answers, LOWER_BOUND, 21)
      ..........
CONTAINS
   SUBROUTINE  First(x, y, z, Lower, LL)
      IMPLICIT  NONE
      INTEGER, INTENT(IN)                     :: Lower
      INTEGER, INTENT(IN)                     :: LL
      INTEGER, DIMENSION(Lower:), INTENT(IN) :: x
      REAL, DIMENSION(1:), INTENT(OUT)       :: y
      LOGICAL, DIMENSION(LL:), INTENT(INOUT) :: z
            ..........
   END SUBROUTINE  First
END PROGRAM   Example
```

Example 2

```
PROGRAM  Test
   IMPLICIT  NONE
   INTEGER, PARAMETER :: MAX_SIZE = 1000
   REAL, DIMENSION(1:MAX_SIZE) :: Data
   INTEGER                     :: ActualSize
   INTEGER                     :: i

   READ(*,*) ActualSize
   READ(*,*) (Data(i), i=1, ActualSize)
   WRITE(*,*) "Sum = ", Sum(Data, ActualSize)

CONTAINS
   REAL FUNCTION  Sum(x, n)
      IMPLICIT  NONE
      REAL, INTENT(IN)               :: n
      REAL, DIMENSION(:), INTENT(IN) :: x
      REAL                           :: Total
      INTEGER                        :: i

      Total = 0.0
      DO i = 1, n
         Total = Total + x(i)
      END DO
      Sum = Total
   END FUNCTION  Sum
END PROGRAM  Test
```

Letter Grade Computation – Pass the Extent

Letter grade is determined using the following scale:

$$
\begin{array}{ll}
\textbf{F} & \textbf{if } x < m - 1.5s \\
\textbf{D} & \textbf{if } m - 1.5s \leq x < m - 0.5s \\
\textbf{C} & \textbf{if } m - 0.5s \leq x < m + 0.5s \\
\textbf{B} & \textbf{if } m + 0.5s \leq x < m + 1.5s \\
\textbf{A} & \textbf{if } x \geq m + 1.5s
\end{array}
$$

where m and s are the means and standard deviations defined as follows:

$$
\begin{aligned}
\textbf{Mean} &= \frac{1}{n}\sum_{i=1}^{n} x_i \\
\textbf{Variance} &= \frac{1}{n}\sum_{i=1}^{n} (x_i - \textbf{Mean})^2 \\
\textbf{Standard Deviation} &= \sqrt{\textbf{Variance}}
\end{aligned}
$$

```
PROGRAM  Grading
   IMPLICIT  NONE
   INTEGER, PARAMETER :: MAX_SIZE = 100
   REAL, DIMENSION(1:MAX_SIZE) :: InputData
   INTEGER                      :: ActualSize

   CALL  ReadArray(InputData, MAX_SIZE, ActualSize)
   CALL  DisplayResult(InputData, MAX_SIZE, ActualSize)
```

```
CONTAINS
   SUBROUTINE ReadArray(x, SIZE, n)
      IMPLICIT  NONE
      INTEGER, INTENT(IN)                   :: SIZE
      INTEGER, INTENT(OUT)                  :: n
      REAL, DIMENSION(1:SIZE), INTENT(OUT) :: x
      INTEGER                               :: i

      READ(*,*)  n
      READ(*,*)  (x(i), i = 1, n)
   END SUBROUTINE  ReadArray

   SUBROUTINE  DisplayResult(Data, SIZE, n)
      IMPLICIT  NONE
      INTEGER, INTENT(IN)                  :: SIZE
      INTEGER, INTENT(IN)                  :: n
      REAL, DIMENSION(1:SIZE), INTENT(IN) :: Data
      INTEGER                              :: i
      REAL                                 :: Mean, Var, Std

      CALL  MeanVariance(Data, SIZE, n, Mean, Var, Std)
      WRITE(*,*)  "Grading Report"
      WRITE(*,*)
      DO i = 1, n
         WRITE(*,*) Data(i), "  ", LetterGrade(Data(i), Mean, Std)
      END DO
      WRITE(*,*)
      WRITE(*,*)  "No. of students          = ", n
      WRITE(*,*)  "Class average            = ", Mean
      WRITE(*,*)  "Class variance           = ", Var
      WRITE(*,*)  "Class standard deviation = ", Std
   END SUBROUTINE  DisplayResult
```

```
   CHARACTER FUNCTION  LetterGrade(x, Mean, StdDev)
      IMPLICIT  NONE
      REAL, INTENT(IN) :: x, Mean, StdDev

      IF (x < Mean - 1.5*StdDev) THEN
         LetterGrade = "F"
      ELSE IF (x < Mean - 0.5*StdDev) THEN
         LetterGrade = "D"
      ELSE IF (x < Mean + 0.5*StdDev) THEN
         LetterGrade = "C"
      ELSE IF (x < 1.5*StdDev) THEN
         LetterGrade = "B"
      ELSE
         LetterGrade = "A"
      END IF
   END FUNCTION  LetterGrade

   SUBROUTINE  MeanVariance(Data, SIZE, n, Mean, Variance, StdDev)
      IMPLICIT  NONE
      INTEGER, INTENT(IN)                    :: SIZE
      INTEGER, INTENT(IN)                    :: n
      REAL, DIMENSION(1:SIZE), INTENT(IN) :: Data
      REAL, INTENT(OUT)                      :: Mean, Variance, StdDev
      INTEGER                                :: i

      Mean = 0.0
      DO i = 1, n
         Mean = Mean + Data(i)
      END DO
      Mean = Mean / n

      Variance = 0.0
      DO i = 1, n
         Variance = Variance + (Data(i) - Mean)**2
      END DO
      Variance = Variance / n
      StdDev   = SQRT(Variance)
   END SUBROUTINE  MeanVariance

END PROGRAM  Grading
```

Letter Grade Computation – Assumed-Shape

```
PROGRAM  Grading
   IMPLICIT  NONE

   INTEGER, PARAMETER :: MAX_SIZE = 100
   REAL, DIMENSION(1:MAX_SIZE) :: InputData
   INTEGER                     :: ActualSize

   CALL  ReadArray(InputData, ActualSize)
   CALL  DisplayResult(InputData, ActualSize)

CONTAINS

   SUBROUTINE ReadArray(x, n)
      IMPLICIT  NONE
      INTEGER, INTENT(OUT)             :: n
      REAL, DIMENSION(1:), INTENT(OUT) :: x
      INTEGER                          :: i

      READ(*,*)  n
      READ(*,*)  (x(i), i = 1, n)
   END SUBROUTINE  ReadArray
```

```
SUBROUTINE  DisplayResult(Data, n)
   IMPLICIT  NONE
   INTEGER, INTENT(IN)              :: n
   REAL, DIMENSION(1:), INTENT(IN) :: Data
   INTEGER                          :: i
   REAL                             :: Mean, Var, Std

   CALL  MeanVariance(Data, n, Mean, Var, Std)
   WRITE(*,*)  "Grading Report"
   WRITE(*,*)
   DO i = 1, n
      WRITE(*,*) Data(i), "  ", LetterGrade(Data(i), Mean, Std)
   END DO
   WRITE(*,*)
   WRITE(*,*)  "No. of students           = ", n
   WRITE(*,*)  "Class average              = ", Mean
   WRITE(*,*)  "Class variance             = ", Var
   WRITE(*,*)  "Class standard deviation = ", Std
END SUBROUTINE  DisplayResult

CHARACTER FUNCTION  LetterGrade(x, Mean, StdDev)
   IMPLICIT  NONE
   REAL, INTENT(IN) :: x, Mean, StdDev

   IF (x < Mean - 1.5*StdDev) THEN
      LetterGrade = "F"
   ELSE IF (x < Mean - 0.5*StdDev) THEN
      LetterGrade = "D"
   ELSE IF (x < Mean + 0.5*StdDev) THEN
      LetterGrade = "C"
   ELSE IF (x < 1.5*StdDev) THEN
      LetterGrade = "B"
   ELSE
      LetterGrade = "A"
   END IF
END FUNCTION  LetterGrade
```

```
   SUBROUTINE  MeanVariance(Data, n, Mean, Variance, StdDev)
      IMPLICIT  NONE
      INTEGER, INTENT(IN)             :: n
      REAL, DIMENSION(1:), INTENT(IN) :: Data
      REAL, INTENT(OUT)               :: Mean, Variance, StdDev
      INTEGER                         :: i

      Mean = 0.0
      DO i = 1, n
         Mean = Mean + Data(i)
      END DO
      Mean = Mean / n

      Variance = 0.0
      DO i = 1, n
         Variance = Variance + (Data(i) - Mean)**2
      END DO
      Variance = Variance / n
      StdDev   = SQRT(Variance)
   END SUBROUTINE  MeanVariance

END PROGRAM  Grading
```

Table Look Up

```
PROGRAM  TableLookUp
   IMPLICIT  NONE
   INTEGER, PARAMETER              :: TableSize = 100
   INTEGER, DIMENSION(1:TableSize) :: Table
   INTEGER                         :: ActualSize, Key, Location
   INTEGER                         :: i, end_of_input
   READ(*,*)  ActualSize
   READ(*,*)  (Table(i), i = 1, ActualSize)
   DO
      WRITE(*,*)  "A search key please --> "
      READ(*,*,IOSTAT=end_of_input)  Key
      IF (end_of_input < 0)  EXIT
      Location = LookUp(Table, ActualSize, Key)
      IF (Location > 0) THEN
         WRITE(*,*)  Key, " appears in location ", Location
      ELSE
         WRITE(*,*)  Key, " is not found"
      END IF
   END DO
CONTAINS
   INTEGER FUNCTION  LookUp(x, Size, Data)
      IMPLICIT  NONE
      INTEGER, DIMENSION(1:), INTENT(IN) :: x
      INTEGER, INTENT(IN)                :: Size, Data
      INTEGER                            :: i
      LookUp = 0
      DO i = 1, Size
         IF (x(i) == Data) THEN
            LookUp = i
            EXIT
         END IF
      END DO
   END FUNCTION  LookUp
END PROGRAM  TableLookUp
```

Sorting

```
PROGRAM  Sorting
   IMPLICIT  NONE
   INTEGER, PARAMETER :: MAX_SIZE = 100
   INTEGER, DIMENSION(1:MAX_SIZE) :: InputData
   INTEGER                        :: ActualSize, i

   READ(*,*)  ActualSize, (InputData(i), i = 1, ActualSize)
   WRITE(*,*) "Input Array:"
   WRITE(*,*) (InputData(i), i = 1, ActualSize)

   CALL  Sort(InputData, ActualSize)

   WRITE(*,*)
   WRITE(*,*) "Sorted Array:"
   WRITE(*,*) (InputData(i), i = 1, ActualSize)

CONTAINS

   INTEGER FUNCTION  FindMinimum(x, Start, End)
      IMPLICIT  NONE
      INTEGER, DIMENSION(1:), INTENT(IN) :: x
      INTEGER, INTENT(IN)                :: Start, End
      INTEGER                            :: Minimum, Location, i
      Minimum  = x(Start)
      Location = Start
      DO i = Start+1, End
         IF (x(i) < Minimum) THEN
            Minimum  = x(i)
            Location = i
         END IF
      END DO
      FindMinimum = Location
   END FUNCTION  FindMinimum
```

```
   SUBROUTINE  Swap(a, b)
      IMPLICIT  NONE
      INTEGER, INTENT(INOUT) :: a, b
      INTEGER                :: Temp
      Temp = a
      a    = b
      b    = Temp
   END SUBROUTINE  Swap

   SUBROUTINE  Sort(x, Size)
      IMPLICIT  NONE
      INTEGER, DIMENSION(1:), INTENT(INOUT) :: x
      INTEGER, INTENT(IN)                   :: Size
      INTEGER                               :: i, Location
      DO i = 1, Size-1
         Location = FindMinimum(x, i, Size)
         CALL  Swap(x(i), x(Location))
      END DO
   END SUBROUTINE  Sort
END PROGRAM  Sorting
```