Chapter 8
Arrays

In many problems, we need to treat a sequence of numbers. For example, we have numbers: $34, 12, 6, 34, \ldots$ and we want to sort them is increasing order.

In linear algebra, we need to use matrix to do calculations.

$$\begin{pmatrix} 1 & 12 & 3 & 0 \\ 0 & 16 & 9 & 36 \\ 12 & 0 & 34 & 9 \\ 2 & 9 & 0 & 11 \end{pmatrix}$$

So we need some data structures to store those data.

One dimensional array

```
REAL, DIMENSION(50) ::  FailureTime
```

This statement tells compiler to establish an array named `FailureTime` consisting of 50 memory locations to store 50 real numbers.

The array can be refereed by the name and the elements of the array (the numbers stored in the array) can be referred by a subscript (or index):

```
FailureTime(1), FailureTime(2), ...  FailureTime(50)
```

```
DO I = 1, 50
  READ (10,*) FailureTime(I)
END DO
```

```
IMPLICIT NONE
INTEGER, PARAMETER :: NumTimes = 50
REAL, DIMENSION(NumTimes) :: FailureTime
... ...
OPEN (UNIT = 10, FILE = "fil8-1.dat", STATUS = "OLD", &
    IOSTAT = OpenStatus)
IF(OpenStatus > 0) STOP "** Cannot open the file **"
READ (10, *, IOSTAT = InputStatus) FailureTime
IF (InputStatus > 0) STOP "** Input error**"
IF (InputStatus < 0) STOP "**Not enough data**"
Sum = 0.0
DO I =1, NumTimes
  Sum = Sum + FailureTime(I)
END DO
... ...
```

Use implied `DO` loop for array input/output

`(list-of-variables, control-var = init-value, limit, step)`

For examples:

```
READ (10,*) (FailureTime(I), I = 1, NumTimes)
```

```
PRINT *, (FailureTime(I), I = 1, NumTimes)
```

Compile-Time arrays: memory of the array is allocated at compile time.

```
type, DIMENSION(low:upper) ::  list-of array-names
```

```
type ::  list-of-array-specifiers
```

where `list-of-array-specifiers` has the form `array-name(low:upper)`.

For examples:

```
INTEGER, DIMENSION(-1:5) ::  Gamma
```

```
REAL ::  Delta(0:49)
```

Compile-time arrays have some disadvantages.

Allocatable arrays (run-time arrays): memory for the array is allocated during execution instead of during compilation.

```
type, DIMENSION(:), ALLOCATABLE ::  list-of-array-names
```

Example: `REAL, DIMENSION(:), ALLOCATABLE ::  A, B`

The actually bound of an allocatable array are specified in an allocate statement.

```
ALLOCATE (array-name(low:upper), STAT = status-variable)
```

The `status-variable` will be set to zero if allocation is successful. A nonzero value indicates an error of allocation.

```
WRITE(*, '(1X, A)', ADVANCE = "NO") &
  "Enter size of arrays A and B:"
READ *, N
ALLOCATE(A(N), B(0:N+1), STAT = AllocateStatus)
IF (AllocateStatus /= 0) STOP "** Not enough memory **"
```

Memory that is no longer needed can be released by a `DEALLOCATE` statement:

```
DEALLOCATE (list-of-arrays, STAT = status-variable)
```

```
PROGRAM Processing_Failure_Time_2
IMPLICIT NONE
REAL, DIMENSION(:), ALLOCATABLE :: FailureTime
INTEGER :: NumTimes
... ...
OPEN (UNIT = 10, FILE = "fil8-1.dat", STATUS = "OLD",&
    IOSTAT = OpenStatus)
IF(OpenStatus > 0) STOP "** Cannot open the file **"
READ (10,*, IOSTAT = InputStatus) NumTimes
... ...
ALLOCATE(FailureTime(NumTimes), STAT = AllocateStatus)
IF (AllocateStatus /= 0) STOP "** Not enough memory**"
READ (10, *, IOSTAT = InputStatus) FailureTime
... ...
DEALLOCATE(FailureTime)
END PROGRAM Processing_Failure_Times_2
```

An array constant may be constructed as a list of values enclosed between (/ and /).

Example: Suppose we defined `INTEGER, DIMENSION(10) :: A`. Then we use any of the assignments:

```
A = (/ 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 /)
A = (/ (2*I, I= 1, 10) /)
A = (/ 2, 4, (I, I = 6, 18, 2), 20 /)
```

Each of the above has the same effect as the following statements.

```
DO I = 1, 10
   A(I) = 2*I
END DO
```

Many operations and functions can be applied to arrays having the same number of elements. Operations applied to an array are carried out elementwise.

```
INTEGER, DIMENSION(4) :: A, B
INTEGER, DIMENSION(0:3) :: C
INTEGER, DIMENSION(6:9) :: D
LOGICAL, DIMENSION(4) :: P
A = (/ 1, 2, 3, 4/)
B = (/ 5, 6, 7, 8/)
C = (/ -1, 3, -5, 7 /)
A = A + B
D = 2 * ABS(C) + 1
P = (C > 0) .AND. (MOD(B,3) == 0)
```

The values of D are: 3, 7, 11, 15 and the values of P are: F T F F.

All the elements in an array can be assigned a simple value:

`A = 0` gives value 0 to each element of A.

Subarrays can use the subscript triples:

`array-name(lower : upper : stride)` specifies the elements in positions `lower, lower+stride, lower+2*stride, ...` going as far as possible without going beyond `upper`. (If stride is negative, then positions going to other way).

When some value in the subscript triple is missing, a default value is used.

A vector subscript can also be used to create a subarray.

```
INTEGER, DIMENSION(10) :: A
INTEGER, DIMENSION(5) :: B, C, D, I

A = (/ 11,22,33,44,55,66,77,88,99, 110 /)
B = A(2:10:2)
I = (/ 6, 5, 3, 9, 1 /)
C = A(I)
D = A((/ 5, 3, 3, 4, 3 /)
```

Then the values are as follows:

B: 22, 44, 66, 88, 110

C: 66, 55, 33, 99, 11

D: 55, 33, 33, 44, 33

```
INTEGER, DIMENSION(10) :: A
A = (/ 11,22,33,44,55,66,77,88,99, 110 /)
A(1:10:2) = (/ I**2, (I = 1, 5) /)
```

Then the values of A are: 1, 22, 4, 44, 9, 66, 16, 88, 25, 110.

Subarray input and output:

`READ(10, *) FailureTime(1:NumTimes)` is same as

```
READ(10, *) (FailureTime(I), I = 1, NumTimes)
```

`PRINT *, FailureTime(1:NumTimes)` is same as

```
PRINT *, (FailureTime(I), I = 1, NumTimes)
```

Using `WHERE` construct to assign values to arrays.

```
INTEGER, DIMENSION(5) :: A=(/ 0,2,5,0,10 /)
REAL, DIMENSION(5) :: B

WHERE (A > 0)
  B = 1.0 /REAL(A)
ELSEWHERE
  B = -1.0
END WHERE
```

The values of B are: $-1.0, 0.5, 0.2, -1.0, 0.1$.

```
WHERE(A /= 0.0)
  A = 1.0 / A ! avoid divide by 0
ELSEWHERE
 A = 9999999.0
END WHERE
```

or

```
WHERE(A /= 0.0)
  A = 1.0 / A
END WHERE
```

The 0 elements in A is unchanged.

Arrays can be used as arguments.

Intrinsic array-processing subprograms:

```
ALLOCATED(A)
DOT_PRODUCT(A, B)
MAXVAL(A)
MAXLOC(A)
MINVAL(A)
MINLOC(A)
PRODENT(A)
SIZE(A)
SUM(A)
```

In programmer-defined subprogram, arrays can also be used as arguments. In this case, formal array arguments must be declared in the subprogram. The actual array arguments must be defined in the calling program unit.

```
FUNCTION Mean(X, NumElements)
 INTEGER, INTENT(IN) :: NumElements
 REAL, DIMENSION(NumElements), INTENT(IN) :: X
 REAL :: Mean
 Mean = SUM(X) /REAL(NumElements)
END FUNCTION Mean
```

To use the Mean function.

```
INTEGER, PARAMETER :: NumItems = 10
REAL, DIMENSION(NumItems) :: Item
PRINT*, "Enter the", NumItems, "real numbers:"
READ *, Item
PRINT '(1X, "Mean of the ", I3, "Numbers is ", F6.2)',&
  NumItems, Mean(Item, NumItems)
```

Or use a run-time array. However, a formal array argument may not be an allocatable array.

```
INTEGER :: NumItems, AllocateStatus
REAL, DIMENSION(:), ALLOCATABLE :: Item
REAL :: Mean
PRINT *, "How many numbers are in the data set?"
READ *, NumItems
ALLOCATE(Item(NumItems), STAT = AllocateStatus)
IF (AllocateStatus /= 0) STOP "**Not enough memory**"
PRINT*, "Enter the", NumItems, "real numbers:"
READ *, Item
PRINT '(1X, "Mean of the ", I3, "Numbers is ", F6.2)',&
  NumItems, Mean(Item, NumItems)
```

Although the formal array argument cannot be an allocatable array, some methods can be used to make subprograms more flexible.

Assumed-shape arrays uses the following declaration.

`DIMENSION(:)` or `DIMENSION(lower:)`

```
INTERFACE
 FUNCTION Mean(X)
  REAL :: Mean
  REAL, DIMENSION(:), INTENT(IN) :: X
  END FUNCTION Mean
END INTERFACE
```

In above example, `X` is not an allocatable array. The size of the array is decided by the actual array.

When a subprogram uses assumed-shape arrays, it may also need some local array variables which size is determined by the assumed-shape array.

```
SUBROUTINE Swap(A, B)
  REAL, DIMENSION(:), INTENT(INOUT) :: A, B
  REAL, DIMENSION(SIZE(A)) :: Temp
  Temp = A
  A = B
  B = Temp
END SUBROUTINE Swap
```

A program-defined function may also return arrays.

Sorting and searching algorithms are common useful algorithms having a lot of applications.

Sorting is used to rearrange the items in a list in either ascending or descending order. There are many sorting algorithms.

Selection sort.

Basic idea: scan the array to find the minimum (maximum) value. Then exchange the values of first element and the minimum (maximun) value. Next scan the elements from second position of the array and do the similar thing $\cdots$.

We can use intrinsic functions `MINVAL, MINLOC` for the program.

```
SUBROUTINE SelectionSort(Item)
 INTEGER, DIMENSION(:), INTENT(INOUT) :: Item
 INTEGER :: NumItems, SmallestItem, LocationSmallest, I
 INTEGER, DIMENSION(1) :: MINLOC_array
 NumItems = SIZE(Item)
 DO I = 1, NumItems - 1
  SmallestItem = MINVAL(Item(I:NumItems))
  MINLOC_array = MINLOC(Item(I:NumItems))
  LocationSmallest = (I - 1) + MINLOC_array(1)
  Item(LocationSmallest) = Item(i)
  Item(I) = SmallestItem
 END DO
END SUBROUTINE SelectionSort
```

Quicksort algorithm is more efficient.

The basic idea: select an element as a pivot, then move all the elements smaller than the pivot to the left of the pivot. Then use a recursive method to do the sorting.

To put the pivot to the position, search from the left to find as element greater than the pivot and search from the right to fine an element less than the pivot. Then interchange these two values. Continue to do that until the search meet at a point and change that point with the pivot.

Example: 50, 30, 20, 80, 90, 70, 95, 85, 10, 15, 75, 25.

The recursive method is as follows.

If the list has only one element or empty, then the sort is done. Otherwise, select a pivot and perform the sequence exchanges described above. Then

1. Splitting the list into two sublist.
2. Recursively sorting the left sublist, and
3. Recursively sorting the right sublist.

```
SUBROUTINE Split(Item, Low, High, Mid)
INTEGER, DIMENSION(:), INTENT(INOUT) :: Item
INTEGER, INTENT(IN) :: Low, High
INTEGER, INTENT(OUT) :: Mid
INTEGER :: Left, Right, Swap
Left = Low
Right = High
Pivot = Item(Low)
DO
  IF(Left >= Right) EXIT
  DO
   IF(Left >= Right .OR. Item(Right) < Pivot) EXIT
   Right = Right - 1
  END DO
  DO
   IF(Item(Left) > Pivot) EXIT
```

```
    Left = Left + 1
  END DO
  IF(Left < Right) THEN
    Swap = Item(Left)
    Item(Left) = Item(Right)
    Item(Right) = Swap
  END IF
END DO
Item(Low) = Item(Right)
Item(Right) = Pivot
Mid = Right
END SUBROUTINE Split
```

```
RECURSIVE SUBROUTINE Quicksort(Item, First, Last)
INTEGER, DIMENSION(:), INTENT(INOUT) :: Item
INTEGER, INTENT(IN) :: First, last
INTEGER :: Mid
 IF(First < Last) THEN
  CALL Split(Item, First, Last, Mid)
  CALL Quicksort(Item, First, Mid-1)
  CALL Quicksort(Item, Mid+1, Last)
 END IF
END SUBROUTINE Quicksort
```

To search for an item in a list is also useful algorithm.

Linear search is simple: compare the searching value to each of the items in the list until find one matching value.

However, if the list is sorted, then we can use a more efficient search algorithm: binary search. The basic idea of the binary search is comparing the searching value with the value stored at the middle of the list. Then

- if two values are the same, the search stops (found).
- if the searching value is less than the value of middle item, then search the first half list.
- if the searching value is greater than the value of middle item, then search the second half list.

```
First = 1
Last = SIZE(Item)
Found = .FALSE.
DO
 IF((First > Last).OR.Found) RETURN
 Middle = (First + Last)/2
 IF(ItemSought < Item(Middle)) THEN
  Last = Middle -1
 ELSE IF (ItemSought > Item(Middle)) THEN
  First = Middle + 1
 ELSE
  Found = .TRUE.
  Location = Middle
 END IF
END DO
```

Multidimensional arrays

For a two dimensional array, we can declare as

```
REAL, DIMENSION(4,3) ::  MatrixA
```

or

```
REAL, DIMENSION(1:4,1:3) ::  MatrixA
```

To refer an element of MatrixA, we can use `MatrixA(i,j)`.

```
DO I = 1, 4
  DO J = 1, 3
   MatrixA(I,J) = I*J
  END DO
END DO
```

Suppose that the temperature readings are made for one week, so that seven temperature tables are collect. Each table recorded one day's temperatures for different time and location:

Sun.	Location		
Time	1	2	3
1	17.5	16.0	15
2	18.5	17.0	16.5
3	16.0	15.5	15.4
4	15.5	14.0	13.8

We can use a three-dimensional array to store these data.

`REAL, DIMENSION(4, 3, 7) :: TemperatureArray` or

`REAL, DIMENSION(1:4, 1:3, 1:7) :: TemperatureArray`

In general, for a k-dimensional array, we can define it as a compile-time array as

```
type, DIMENSION(l1:u1, l2:u2, ...  lk:uk) ::  &
     name1, name2, ...
```

We can also declare it as a run-time array:

`type, DIMENSION(:, :, ..., :), ALLOCABLE ::  name1, name2, ...` and use `ALLOCATE` statement to specify the size of the array at run-time.

```
REAL, DIMENSION(:, :, :), ALLOCABLE :: Beta
REAL, DIMENSION(:, :), ALLOCABLE :: Gamma
INTEGER :: AllocateStatus, I, J, K
... ...
ALLOCATE(Beta(0:2, 0:3, 1:2), Gamma(1:2, -1:3),&
   STAT = AllocateStatus)
... ...
DO I = 0, 2
 DO J = 0, 3
  DO K = 1, 2
   Beta(I, J, K) = I + J + K
  END DO
 END DO
END DO
READ *, ((Gamma(I,J), J = -1, 3), I = 1, 2)
```

```
REAL, DIMENSION(:, :), ALLOCATABLE :: Temperature
INTEGER :: NumTimes, NumLocs, AllocateStatus,Time, Location
PRINT *, "Enter number of times temperatures are recorded"
PRINT *, "and number of locations where recorded:"
READ *, NumTimes, NumLocs
ALLOCATE(Temperature(NumTimes, NumLocs), STAT = AllocateStatus)
IF (AllocateStatus /= 0) STOP "**Not enough memory**"
PRINT *, "Enter the temperatures at the first location,"
PRINT *, "then those at the second location, and so on:"
READ *, ((Temperature(Time, Location), &
  Location = 1, NumLocs), Time =1, NumTimes)
... ...
DEALLOCATE(Temperatures)
```