Computer Science 1510

Lecture 12

February 1, 2016

Lecture Outline

- Arrays

Structured data

- So far, all the data elements (variables) we have used in our programs have had distinct, individual names. (Preferably, names that represent something in the problem being solved.)

- Often, though, we are interested in collections of data, where the data elements are related to each other.

- We often call such collections of related data a data structure.

- Perhaps the simplest data structure is a list of similar data elements.

- Examples of lists could be a list of course grades, a set of telephone numbers, the hourly temperature measurements for a day, or the components of a vector.

Lists

- Lists are a convenient way of organizing similar data elements.

- Typically, we give the list a name, and identify individual data elements in the list by their location in the list.

- The location is an index pointing to the particular data element to which it refers.

- For example, for a list with elements (A1, A2, A3, A4) the index 3 refers to the third element in the list with name (or label) A.

- We often use subscripts to identify individual elements in a list.

  In fact, the memory of a computer is typically organized as one large list. A memory address is simply an index, and the list element is the data stored at that memory location.

Arrays

- Fortran has, as part of its language, the concept of an array.
- Lists are simply a 1-dimensional array.
- Arrays are sequential and contiguous sections of memory that can be allocated by a program.
- Each element of an array has the same type (INTEGER, REAL, etc.).
- The array has a single variable name.
- A 1-dimensional array has an index, or subscript, which refers to individual elements in the array.

Declaration of a one-dimensional array

- Syntax:

  `type,DIMENSION(start:end)::array_name`

  or

  `type::array_name(start:end)`

  where `type` is any Fortran data type, `array_name` is any legal Fortran identifier, `start` is the initial index, and `end` is the final index. If `start=1` then only `end` has to be specified.

- For example, `INTEGER::grades(8)` creates a one-dimensional array of integers called `grades` which has 8 elements.

- `grades` can now store 8 separate integers.

- To access a particular element in the array we would specify its position in the array, called its *index*.

- Each element in the array has a unique index number. For example, `grades(3)` refers to the third element in the array.

Using arrays

- Array elements can be used in any location where a single variable can be used. For example,
  - An array element can be assigned a value,
    ```
    array_name(index)=expression
    ```
  - A variable can be assigned the value of an array element,
    ```
    variable=array_name(index)
    ```
  - The value of an array element can be displayed,
    ```
    WRITE(*,*) array_name(index)
    ```
  - An array element can be used in an arithmetic or logical expression,
    ```
    X=array_name(index)**2
    IF(array_name(index)==0) EXIT
    ```

- Note that in each case above, the `index` can be an integer constant, an integer variable, or an expression that evaluates to an integer.

Array initialization

- It is always good practice to initialize your variables.
- Array initialization can be done using a loop. For example,

```
DO i=1,8
   grades(i)=0
END DO
```

- Notice the use of the loop counter to index into the array. The initialization statement is executed with a different value of `i` on each iteration of the loop (from `i=1` up to `8`).
- The size of the array is often referred to as the *dimension* of the array.
- If all elements of an array are to be initialized to the same value, a single assignment statement can be used,

```
grades = 0
```

Array input/output

- One method of reading in values for an array is to use a loop,

```
DO i=1,8
   READ(*,*) grades(i)
END DO
```

- This is equivalent to 8 READ statements.

- Alternatively, the single READ statement,

```
READ(*,*) grades
```

  can be used. Execution will be suspended until enough values to fill the array have been entered.

- An *implied DO loop* can also be used,

```
READ(*,*) (grades(i), i=1,8)
```

- The above methods also apply to array output (ie. WRITE statements).

Example 1: Arrays

```
PROGRAM Processing_Failure_Times_1
!-------------------------------------------------------------------------
! Program to read a list of failure times, calculate the mean time to
! failure, and print a list of failure times that are greater than the mean.
!   Input:
!     FailureTime          : one-dimensional array of failure times
!     NumTimes             : size of the array (constant)
!   Output:
!     Mean_Time_to_Failure : mean of the failure times
!     List of failure times greater than Mean_Time_to_Failure
!-------------------------------------------------------------------------
  IMPLICIT NONE
  INTEGER,PARAMETER::NumTimes=50
  REAL,DIMENSION(NumTimes)::FailureTime
  INTEGER::I
  REAL::Sum,Mean_Time_to_Failure

  ! Read the failure times and store them in array FailureTime
  WRITE(*,*) 'Enter the ', NumTimes, 'failure times'
  READ(*,*) FailureTime

  ! Calculate the mean time to failure
  Sum=0.0
  DO I=1,NumTimes
     Sum=Sum+FailureTime(I)
  END DO
  Mean_Time_to_Failure = Sum/NumTimes
  WRITE(*,'(A,X,F6.1)') 'Mean time to failure =', Mean_Time_to_Failure

  ! Print list of failure times greater than the mean
  WRITE(*,*)  'List of failure times greater than the mean:'
  DO I=1,NumTimes
     IF (FailureTime(I) > Mean_Time_to_Failure) &
        WRITE(*,'(F9.1)') FailureTime(I)
  END DO
END PROGRAM Processing_Failure_Times_1
```

Example 2: Arrays

```
PROGRAM Grade_array
!-------------------------------------------------------------
! This program accepts assignment marks from the user, and
! calculates and displays the average assignment mark, as
! well as the maximum and minimum mark.
!  INPUT:
!    n - number of assignment marks to be entered.
!    assign - array of assignment marks (of size n).
!  OUTPUT:
!    assign_avg - average of the assignment marks.
!    assign_max - maximum assignment mark
!    assign_min - minimum assignment mark
!-------------------------------------------------------------
   IMPLICIT NONE
   INTEGER::assign(999),i,n
   REAL::assign_avg
   INTEGER::assign_max,assign_min
   INTERFACE
      FUNCTION Avg(marks,n)
         INTEGER,INTENT(IN)::marks(n),n
         REAL::Avg
      END FUNCTION Avg
      SUBROUTINE Maxmin(marks,n,amax,amin)
         INTEGER,INTENT(IN)::marks(:),n
         INTEGER,INTENT(OUT)::amax,amin
      END SUBROUTINE Maxmin
   END INTERFACE

   WRITE(*,*) 'How many assignment grades would you like to enter?'
   READ(*,*) n
   DO i=1,n
      WRITE(*,10) 'Enter assignment mark ',i
      READ(*,*) assign(i)
   END DO
10 FORMAT(A25,I1)
```

```
   assign_avg=Avg(assign,n)
   WRITE(*,*) 'The average assignment mark is ',assign_avg

   CALL Maxmin(assign,n,assign_max,assign_min)
   WRITE(*,*) 'The maximum assignment mark is ',assign_max
   WRITE(*,*) 'The minimum assignment mark is ',assign_min
END PROGRAM Grade_array

FUNCTION Avg(marks,n)
   INTEGER,INTENT(IN)::marks(n),n
   REAL::Avg
   INTEGER::i  !  This is a local variable
   REAL::sum ! This is a local variable
   sum=0.0
   DO i=1,n
      sum=sum+marks(i)
   END DO
   Avg=sum/n
   RETURN
END FUNCTION Avg

SUBROUTINE Maxmin(marks,n,amax,amin)
   INTEGER,INTENT(IN)::marks(:),n
   INTEGER,INTENT(OUT)::amax,amin
   INTEGER::i
   amax=marks(1)
   amin=marks(1)
   DO i=2,n
      IF(marks(i)>amax) THEN
           amax=marks(i)
      END IF
      IF(marks(i)<amin) THEN
           amin=marks(i)
      END IF
   END DO
   RETURN
END SUBROUTINE Maxmin
```

Notes on PROGRAM Grade_array

- The above program is an example of using a combination of functions and subroutines within the same program.

- We have declared an array with a size much larger than required such that the user can choose how many marks to enter. This makes the program much more general.

- The user supplied number of marks is passed into the function `Avg` and used to determine the number of loop iterations.

- In the subroutine `Maxmin` we have used an *assumed size array* (`:` as the array size), where the subroutine assumes the same size as in the main program.

- With the function and subroutine written in this way, both can be easily reused in another application where the specific tasks are required.

Array processing

- Consider an integer array of 6 elements called `A`. Array elements can be assigned values in any of the following ways:

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

- Operators and functions that we have so far applied to simple variables expressions can also be applied to arrays. Each operation is performed element-wise. For example,

```
INTEGER::A(4)=(/ 1,2,3,4 /)
INTEGER::B(4)=(/ 5,6,7,8 /)
INTEGER::C(4)
C=A+B
```

  will result in the elements of `C` having values 6,8,10,12.

Array processing

- Logical expressions can also be used. For example,

```
P = (C > 0)
```

- Portions of an array can be extracted using an *index triplet*. For example,

```
A(2:10:2)
```

  refers to elements 2, 4, 6, 8, and 10 of the array `A`.

- An *index vector* can also be used to extract elements of an array. For example,

```
INTEGER::A(6) = (/ 2, 4, 6, 8, 10, 12 /)
INTEGER::B(3),I(3) = (/ 4, 1, 2 /)
B=A(I)
```

  results in an array `B` with elements 8, 2, 4.

Array processing

There are several built-in array processing subprograms in Fortran. The following is a partial list.

`ALLOCATED(A)`	Returns true if memory has been allocated for `A` (see later).
`DOT_PRODUCT(A,B)`	Returns the dot product of arrays `A` and `B`.
`MAXVAL(A)`	Returns the maximum value in array `A`.
`MAXLOC(A)`	Returns the location of the first occurrence of the maximum value in array `A`.
`MINVAL(A)`	Returns the minimum value in array `A`.
`MINLOC(A)`	Returns the location of the first occurrence of the minimum value in array `A`.
`PRODUCT(A)`	Returns the product of the elements in `A`.
`SIZE(A)`	Returns the number of elements in `A`.
`SUM(A)`	Returns the sum of the elements in `A`.