Computer Science 1510

Lecture 18

Lecture Outline

- Other data types
- Multidimensional arrays
- Allocatable arrays

Fortran data type: COMPLEX

- Recall that, in addition to the INTEGER, REAL, CHARACTER, and LOGICAL data types that we have been using, there is also a COMPLEX data type in Fortran.

- The COMPLEX data type is used to represent complex numbers, ie. numbers of the form,

  $$z = a + bi$$

  where a is the *real part* and b is the *imaginary* or *complex part*.

- In Fortran, a complex number is represented as an ordered pair (a, b).

- Syntax:

  `COMPLEX::identifier-list`

  where `identifier-list` is a list of variables to be declared of type COMPLEX.

Fortran data type: COMPLEX

- It is also possible to have arrays of COMPLEX values.

- There is a set of Fortran library functions that can be used with COMPLEX variables. The following example will illustrate a few.

- Example:

```
PROGRAM Test_complex
   IMPLICIT NONE
   COMPLEX::z
   REAL::x,y,mag
   z=(2.0,3.0)  ! set z=2+3i
   mag=CABS(z)  ! compute the magnitude of z (as a real number)
   x=REAL(z)    ! extract the real part of z
   y=AIMAG(z)   ! extract the imaginary part of z
   WRITE(*,5) 'z=',z,'mag=',mag,'x=',x,'y=',y
5  FORMAT(A,F4.1,'+',F4.1,'i',A,F4.1,A,F4.1,A,F4.1)
END PROGRAM Test_complex
```

- Output

```
z= 2.0+ 3.0i, mag= 3.6, x= 2.0, y= 3.0
```

Fortran: KIND

- So far, for real number data we have used the REAL data type. Variables declared as REAL usually occupy 32 bits in memory (ie. single precision).

- The number of bits used for a particular representation restricts the range and precision of the numbers that can be stored.

- If higher precision or greater range is required, we set the KIND of the variable.

- The KIND value for single precision variables is either KIND=1 or KIND=4 depending on the type of computer. For us it is 4.

- For example, we can obtain 15 decimal place precision, and a of range $[10^{-37}, 10^{37}]$) by:

```
INTEGER, PARAMETER :: dp1 = selected_real_kind(15, 307)
REAL(KIND=dp1) :: b
```

- The KIND statement can also be used for integers.

Fortran: KIND example

```
PROGRAM Test_kind
   USE, INTRINSIC :: iso_fortran_env
   IMPLICIT NONE
   REAL :: x
   INTEGER, PARAMETER :: sp = selected_real_kind(6, 37)
   INTEGER, PARAMETER :: dp1 = selected_real_kind(15, 307)
   INTEGER, PARAMETER :: dp2 = kind(1.d0)
   INTEGER, PARAMETER :: sp_mod = REAL32
   INTEGER, PARAMETER :: dp_mod = REAL64
   REAL(KIND=sp)  :: a
   REAL(KIND=dp1) :: b
   REAL(dp2)      :: c
   REAL(sp_mod)   :: d
   REAL(dp_mod)   :: e
   WRITE(*,*) 'The largest value for x is', HUGE(x)
   WRITE(*,*) 'sp = ',sp
   WRITE(*,*) 'The largest value for a is', HUGE(a)
   WRITE(*,*) 'dp1 = ',dp1
   WRITE(*,*) 'The largest value for b is', HUGE(b)
   WRITE(*,*) 'dp2 = ',dp2
   WRITE(*,*) 'The largest value for c is', HUGE(c)
   WRITE(*,*) 'sp_mod = ',sp_mod
   WRITE(*,*) 'The largest value for d is', HUGE(d)
   WRITE(*,*) 'dp_mod = ',dp_mod
   WRITE(*,*) 'The largest value for e is', HUGE(e)
   WRITE(*,*) 'The precision of e is',PRECISION(e)
   WRITE(*,*) 'The range of e is',RANGE(e)
END PROGRAM Test_kind
```

Fortran: KIND example

Output:

```
The largest value for x is   3.40282347E+38
sp =            4
The largest value for a is   3.40282347E+38
dp1 =            8
The largest value for b is   1.7976931348623157E+308
dp2 =            8
The largest value for c is   1.7976931348623157E+308
sp_mod =            4
The largest value for d is   3.40282347E+38
dp_mod =            8
The largest value for e is   1.7976931348623157E+308
The smallest value for e is   2.2250738585072014E-308
The precision of e is          15
The range of e is         307
```

Note the use of the `HUGE` function. There is also a `TINY` function that will determine the smallest number possible for a given type/kind.

Multidimensional arrays

- The arrays that we have used thus far have had a single subscript.

- However, it is possible in Fortran to have multi-dimensional arrays, in which case an element of the array is referred to by attaching the appropriate number of indices to the array name.

- Syntax:

```
type, DIMENSION(l1:u1, l2:u2, ..., lk:uk) :: array_name
                              or
     type :: array_name(l1:u1, l2:u2, ..., lk:uk)
```

  where `k` is the number of dimensions (or *rank*) of the array (at most 15), and `li/ui` specify the lower and upper indices for each dimension.

Allocatable arrays

- The arrays that we have considered thus far have had their memory allocated at compile time since the array size was set when the program was compiled.

- In the case where we do not know ahead of time how much space is required it would be more efficient to be able to allocate the memory at run time.

- For example, if the user is asked to enter the number of elements, with compile-time arrays we had to declare an array much larger than required to ensure that we would have enough space.

Allocatable arrays

- In order to allocate an array during run-time we first have to declare it as `ALLOCATABLE`.
- Syntax:

```
type, DIMENSION(:), ALLOCATABLE :: array_name
```

  where the `:` within the `DIMENSION` clause indicates that the size is not yet defined, and the `ALLOCATABLE` clause indicates that the size will be specified during run-time.

- To allocate memory for such arrays we use the `ALLOCATE` statement.
- Syntax:

```
ALLOCATE(array_name(start:end),STAT=status-variable)
```

  where `status-variable` is an integer variable whose value will be zero if the allocation was successful, and nonzero if there was insufficient memory or if the array has already been allocated.

Allocatable arrays

- The following code allocates space for `n` integers, where `n` is input by the user.

```
INTEGER :: array_status,n
INTEGER, DIMENSION(:), ALLOCATABLE :: myarray
WRITE(*,*) 'How many elements are required?'
READ(*,*) n
ALLOCATE(myarray(n),STAT=array_status)
IF (array_status /= 0) THEN
   WRITE(*,*) 'Unable to allocate'
   STOP
END IF
```

- Following the `ALLOCATE` statement, allocatable (run-time) arrays can be used in the same way as arrays allocated at compile-time.

Allocatable arrays

- The `ALLOCATE` statement requests from the operating system, a block of memory of the required size. Provided that there is sufficient memory for this request the memory is assigned to the program.

- When the memory assigned to the program is no longer needed, it should be set free (back to the operating system). To do this the `DEALLOCATE` statement can be used.

- Syntax:

  ```
  DEALLOCATE(array_name,STAT=status-variable)
  ```

  where `status-variable` will be set to zero if the deallocation is successful, and nonzero if it is not. The latter case can arise, for example, if the memory to be deallocated was never allocated.

Example: Allocatable arrays

```
PROGRAM Processing_Failure_Times_2
!-------------------------------------------------------------------------
! Program to read a list of failure times, calculate the mean time to
! failure, and then print a list of failure times that are greater
! than the mean.  An allocatable array is used to store the failure
! times.  Identifiers used are:
!     FileName             : name of file from which to read times
!     OpenStatus           : status variable for OPEN
!     InputStatus          : status variable for READ
!     AllocateStatus       : status variable for ALLOCATE
!     FailureTime          : one-dimensional array of failure times
!     NumTimes             : size of the array
!     I                    : subscript
!     Sum                  : sum of failure times
!     Mean_Time_to_Failure : mean of the failure times
!
! Input:  FileName
!         File should contain a value for NumTimes following by the
!         failure times.
! Output: Mean_Time_to_Failure and a list of failure times greater
!         than Mean_Time_to_Failure
!-------------------------------------------------------------------------

   IMPLICIT NONE
   REAL, DIMENSION(:), ALLOCATABLE :: FailureTime
   INTEGER :: OpenStatus, InputStatus, AllocateStatus, NumTimes, I
   REAL :: Sum, Mean_Time_to_Failure
   CHARACTER(Len=20) :: FileName

   ! Get the filename from the user
   WRITE(*,'(A)',ADVANCE='NO') 'Please enter the name of the file to be read: '
   READ(*,*) FileName

   ! Open the file
   OPEN(UNIT=10,FILE=FileName,STATUS='OLD',IOSTAT=OpenStatus)
```

```
IF (OpenStatus > 0) THEN
   WRITE(*,*) 'Cannot open file!'
   STOP
END IF

! Get the number of failure times
READ(10,*,IOSTAT=InputStatus) NumTimes
IF (InputStatus > 0) THEN
   WRITE(*,*) 'Input error'
   STOP
ELSE IF (InputStatus < 0) THEN
   WRITE(*,*) 'Not enough data'
   STOP
END IF

! Allocate an array with NumTimes elements to store the failure times
ALLOCATE(FailureTime(NumTimes),STAT=AllocateStatus)
IF (AllocateStatus /= 0) THEN
   WRITE(*,*) 'Unable to allocate memory'
   STOP
END IF

! Read the failure times and store them in array FailureTime
READ(UNIT=10,FMT=*,IOSTAT=InputStatus) FailureTime
IF (InputStatus > 0) THEN
   WRITE(*,*) 'Input error'
   STOP
ELSE IF (InputStatus < 0) THEN
   WRITE(*,*) 'Not enough data'
   STOP
END IF

! Calculate the mean time to failure
Sum = 0.0
DO I = 1, NumTimes
   Sum = Sum + FailureTime(I)
END DO
Mean_Time_to_Failure = Sum / REAL(NumTimes)
```

```
   WRITE(*,'(A,X,F6.1)') 'Mean time to failure =', Mean_Time_to_Failure

   ! Print list of failure times greater than the mean
   WRITE(*,*)
   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

   ! Deallocate the array of failure times
   DEALLOCATE(FailureTime)

END PROGRAM Processing_Failure_Times_2
```

Example: TYPE with ALLOCATE

```
PROGRAM Structure
   IMPLICIT NONE
   ! Define a Student data structure
   TYPE Student
      CHARACTER(Len=9)::id_num
      CHARACTER(Len=15)::first_name
      CHARACTER(Len=20)::last_name
      REAL::assign(8),mid,final,grade
   END TYPE Student
   INTEGER::i,j,num,OpenStatus,InputStatus,AllocateStatus
   REAL::total,aavg
   CHARACTER(Len=20)::FileName

   ! Declare an allocatable array of Students
   TYPE(Student),DIMENSION(:),ALLOCATABLE::Class

   WRITE(*,'(A)',ADVANCE='NO') 'Which file should be read?  '
   READ(*,*) FileName
   OPEN(UNIT=5,FILE=FileName,STATUS='OLD',IOSTAT=OpenStatus)
   IF (OpenStatus > 0) STOP

   ! Determine how many students are in the class
   READ(5,*,IOSTAT=InputStatus) num
   IF (InputStatus /= 0) STOP

   ! Allocate memory for the Class
   ALLOCATE(Class(num),STAT=AllocateStatus)
   IF (AllocateStatus /= 0) STOP

   ! Read in the student data
   DO i=1,num
      READ(5,*,IOSTAT=InputStatus) Class(i)%id_num
      IF (InputStatus /= 0) STOP
      READ(5,*,IOSTAT=InputStatus) Class(i)%first_name
      IF (InputStatus /= 0) STOP
```

```
      READ(5,*,IOSTAT=InputStatus) Class(i)%last_name
      IF (InputStatus /= 0) STOP
      READ(5,*,IOSTAT=InputStatus) Class(i)%assign
      IF (InputStatus /= 0) STOP
      READ(5,*,IOSTAT=InputStatus) Class(i)%mid
      IF (InputStatus /= 0) STOP
      READ(5,*,IOSTAT=InputStatus) Class(i)%final
      IF (InputStatus /= 0) STOP
   END DO

   ! For each Student, compute the final mark
   DO j=1,num
      ! Compute the average assignment mark for Student j
      total=0
      DO i=1,8
         total=total+Class(J)%assign(i)
      END DO
      aavg=total/8.0

      ! Compute the final mark for Student j
      Class(j)%grade=aavg/20.0*30.0+Class(j)%mid*2*0.3+Class(j)%final*2*0.4

      ! Display the final mark for Student j
      WRITE(*,10) Class(j)%first_name,Class(j)%last_name,'received',&
                  Class(j)%grade,'in the course.'
   END DO
10 FORMAT(A15,A20,X,A8,X,F5.1,X,A14)

   DEALLOCATE(Class)
   CLOSE(UNIT=5)
END PROGRAM Structure
```

Allocatable arrays

- We have seen how we can allocate memory during run-time to prevent having to declare more space than is required.

- However, what if we want to change the size of an array during execution?

- With the techniques that we have learned thus far, this would require the following:
  - Allocate a new, larger, array.
  - Copy the elements from the old array to the new array.
  - Write the new elements to the new array.
  - Deallocate the old array.