Computer Science 1510

Lecture 13

February 3, 2016

Lecture Outline

- Searching
- File input/output

Searching

- A common programming problem is searching a list of data for a particular item.

- Assume that the elements in the list are in no particular order.

- To find the required list item, a search could start with the first element in the list and search sequentially until either the desired item is found or the end of the list is reached.

- This is called a linear search.

Example: Linear Search

```
PROGRAM Linsearch
    IMPLICIT NONE
    INTEGER :: list(999), num, i, location, to_find
    LOGICAL :: found=.FALSE.

    WRITE(*,*) 'How many items are to be entered?'
    READ(*,*) num
    WRITE(*,*) 'Enter values'
    READ(*,*) (list(i),i=1,num)
    WRITE(*,*) 'What value do you want to find?'
    READ(*,*) to_find

    location = 1 ! Start with checking the first element
    DO
        ! Terminate if item is found, or end of list has been reached
        IF ((location > num).OR.found) EXIT

        ! Check next element in the list
        IF (list(location) == to_find) THEN
            found = .TRUE.
        ELSE ! Move to next element
            location = location + 1
        END IF
    END DO

    IF (found) THEN
        WRITE(*,'(A,I4,A,I4)') 'Value ', to_find,' was found at location ',location
    ELSE
        WRITE(*,*) 'Item not found'
    END IF

END PROGRAM Linsearch
```

File input/output

- For problems involving large data sets, input from the keyboard and output to the screen are impractical. I/O from/to files is required.

- There are four basic operations associated with file input/output:

  1. Opening the file.
  2. Reading from the file.
  3. Writing to the file.
  4. Closing the file.

Opening a file

- Opening a file allows you to access the file contents, whether it be to read the contents or add (write) additional content.

- Syntax:

```
OPEN (open-list)
```

  where `open-list` can include the following:

  - `UNIT=integer` – Unit number associated with the file, any integer can be used

  - `FILE=string` – Name of the file in quotes

  - `STATUS=string` – Status of the file. Possible values are 'OLD', 'NEW', and 'REPLACE'.

  - `ACTION=string` – How the file will be used. Possible values are 'READ', 'WRITE', and 'READWRITE'.

  - `POSITION=string` – At what position we start in the file. Possible values are 'REWIND', 'APPEND', and 'ASIS'.

  - `IOSTAT=integer_variable` – The value of the `integer_variable` indicates whether the file was opened successfully. A value of zero indicates that the file was opened successfully. A positive value is assigned otherwise.

  - There are other clauses that can be included in the `open-list`.

- Once a file is open, it remains accessible until a `CLOSE` statement is issued.

- Note that unit number 6 refers to standard input (keyboard) and standard output (screen).

Opening a file

- Examples:

  1. Open a file called `data.dat` with unit number 2.

     ```
     OPEN(UNIT=2,FILE='data.dat')
     ```

  2. Open a new file called `names.dat` with unit number 10.

     ```
     OPEN(UNIT=10,FILE='names.dat',STATUS='NEW')
     ```

  3. Open an existing file called `prices.dat` with unit number 5.

     ```
     OPEN(UNIT=5,FILE='prices.dat',STATUS='OLD')
     ```

  4. Open a file with the filename specified by the user and stored in a character array, `openfile`.

     ```
     CHARACTER(20) :: openfile
     READ(*,*) openfile
     OPEN(UNIT=7,FILE=openfile)
     ```

Closing a file

- When you are done accessing the file you should *close* the file.

- Closing a file has the following effects:

  1. Information can no longer be retrieved from or added to the file.
  2. Information still in a RAM buffer is copied into the file.
  3. No data is lost if a program terminates prematurely.

- Syntax: (in its simplest form)

  ```
  CLOSE(UNIT=integer)
  ```

  where the unit number must match that assigned to the file in the OPEN statement.

Generalized READ statement

- Syntax:

```
READ(control-list) input-list
```

  where `input-list` is a variable or list of variables separated by commas, and `control-list` may include the following:

  - A unit specifier, ex. `UNIT=5`.
  - A format specifier, ex. `FMT='(I3)'`
  - An `IOSTAT` clause.
  - A `ADVANCE` clause.
  - There are other clauses that can be included as well.

- For a READ statement the IOSTAT variable has a positive value if an input error occurs, a negative value if the end of the data is encountered but no input error occurs, and zero if successful.

Reading from a file

- To read the contents of a file we use the READ statement that we have been using all along.

- To tell the READ statement to read from the file instead of the keyboard we specify the file unit number in place of the first `*`.

- Examples:

  1. Read a value `n` from *standard input* (the keyboard).

     ```
     READ(*,*) n
     ```

  2. Read a value `x` from a file called `prices.dat`.

     ```
     READ(5,*) x
     ```

- Note that the OPEN statement which defines the unit number must occur before the READ statement that accesses the file.

Reading from a file

- Recall that the second `*` in the READ statement can be changed to an integer identifying a FORMAT statement.

- For example, to read integers `n` and `m` from a file called `prices.dat` we could have the following:

```
      OPEN(UNIT=4,FILE='prices.dat',STATUS='OLD')
      READ(4,5) n,m
5     FORMAT(I3,I3)
```

- The `4` associates the READ statement with the file `prices.dat` which was opened with unit number `4`.

- The `5` associates the READ statement with the FORMAT statement labelled `5` which specifies the type of data to be read from the file. In this case we are expecting two integers up to 3 digits each.

Example 1: File input

```
PROGRAM Processing_Failure_Times_1
!-----------------------------------------------------------------------
! 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.  Identifiers used are:
!     OpenStatus           : status variable for OPEN
!     InputStatus          : status variable for READ
!     FailureTime          : one-dimensional array of failure times
!     NumTimes             : size of the array (constant)
!     I                    : subscript
!     Sum                  : sum of failure times
!     Mean_Time_to_Failure : mean of the failure times
!     FileName             : name of file from which to read times
!
! Input:  Name of file from which to read failure times
! Output: Information to user about the data file,
!         Mean_Time_to_Failure, and a list of failure times greater
!         than Mean_Time_to_Failure
!-----------------------------------------------------------------------
  IMPLICIT NONE
  INTEGER,PARAMETER::NumTimes=50
  REAL,DIMENSION(NumTimes)::FailureTime
  INTEGER::OpenStatus,InputStatus,I
  REAL::Sum,Mean_Time_to_Failure
  CHARACTER(Len=20)::FileName

  WRITE(*,*) 'Which file would you like to read failure times from?'
  READ(*,*) FileName

  ! Open file from which to read the failure times
  OPEN(UNIT=10,FILE=FileName,STATUS='OLD',IOSTAT=OpenStatus)
  ! Check that file was opened successfully
  IF (OpenStatus > 0) THEN
     WRITE(*,*) 'Cannot open file!'
     STOP
  END IF
```

```
  ! Read the failure times and store them in array FailureTime
  READ(UNIT=10,FMT='(F4.1)',IOSTAT=InputStatus) FailureTime
  IF (InputStatus > 0) THEN
     WRITE(*,*) 'Input error'
     STOP
  ELSE IF (InputStatus < 0) THEN
     WRITE(*,*) 'Not enough data'
     STOP
  END IF
  CLOSE(UNIT=10)

  ! 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(*,*)
  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: File input

```
PROGRAM Failure_Times_EOF
!-------------------------------------------------------------------------
! 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.  Failure times are read from a file until the end
! of the file is reached.  Identifiers used are:
!     OpenStatus           : status variable for OPEN
!     InputStatus          : status variable for READ
!     FailureTime          : one-dimensional array of failure times
!     I                    : loop counter
!     NumTimes             : number of times read
!     Sum                  : sum of failure times
!     Mean_Time_to_Failure : mean of the failure times
!     FileName             : name of file from which to read times
!     ValueRead            : value read from file
!
! Input:  Name of file from which to read failure times
! Output: Information to user about the data file,
!         Mean_Time_to_Failure, and a list of failure times greater
!         than Mean_Time_to_Failure
!-------------------------------------------------------------------------
  IMPLICIT NONE
  REAL,DIMENSION(999) :: FailureTime
  INTEGER :: OpenStatus, InputStatus, I, NumTimes
  REAL :: Sum, Mean_Time_to_Failure, ValueRead
  CHARACTER(Len=20) :: FileName

  WRITE(*,*) 'Which file would you like to read failure times from?'
  READ(*,*) FileName

  ! Open file from which to read the failure times
  OPEN(UNIT=10,FILE=FileName,STATUS='OLD',IOSTAT=OpenStatus)
  ! Check that file was opened successfully
  IF (OpenStatus > 0) THEN
     WRITE(*,*) 'Cannot open file!'
     STOP
```

```
  END IF

  I=0
  Sum=0.0
  ! Read the failure times and store them in array FailureTime
  DO
    READ(UNIT=10,FMT='(F4.1)',IOSTAT=InputStatus) ValueRead
    IF (InputStatus > 0) THEN
        WRITE(*,*) 'Input error'
        STOP
    ELSE IF (InputStatus < 0) THEN
        ! Stop reading failure times when end of file is reached.
        WRITE(*,*) 'Reached end of file'
        EXIT
    END IF
    ! Value read successfully, add to array and sum
    I = I+1
    FailureTime(I) = ValueRead
    Sum=Sum+FailureTime(I)
  END DO
  CLOSE(UNIT=10)

  NumTimes = I
  WRITE(*,'(A,I4,A)') 'Read ',NumTimes,' failure times'
  ! Calculate the mean time to failure
  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(*,*)
  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 Failure_Times_EOF
```

Example: File output

```
PROGRAM Trig_table
!-----------------------------------------------------------------
! This program creates a table containing values of sin(x)
! and cos(x), given starting and ending values for x, as well
! as the number of intervals to include in the table.
!  INPUT:
!    minvalue - the minimum value for x.
!    maxvalue - the maximum value for x.
!    nvals - the number of intervals between minvalue & maxvalue
!  OUTPUT:  A file containing x, sin(x), and cos(x)
!-----------------------------------------------------------------
    IMPLICIT NONE
    REAL :: x, init, minvalue, maxvalue, inc
    INTEGER :: nvals, i

    OPEN(UNIT=7,FILE='table.dat',STATUS='REPLACE')

    WRITE(*,*) 'What is the starting value'
    READ(*,*) minvalue
    WRITE(*,*) 'What is the ending value'
    READ(*,*) maxvalue
    WRITE(*,*) 'How many values would you like in the table?'
    READ(*,*) nvals
    ! Write headers to the file
    WRITE(7,*) 'x       sin(x)    cos(x)'
    WRITE(7,*) '------------------------'

    ! Compute the increment
    inc = (maxvalue-minvalue)/(nvals-1)
    x = minvalue
    DO i=1,nvals
        WRITE(7,10) x, sin(x), cos(x)
        x = x + inc
    END DO
10  FORMAT(F6.3,2(3X,F6.3))
END PROGRAM Trig_table
```