Problem

Read 4 test scores (1 per line) and 1) find the average 2) print the average, 3) and finally print the 4 scores; (in this order).

Solution:

```
: READ(9,100) score1
READ(9,100) score2
READ(9,100) score3
READ(9,100) score4
ave = ( score1 + score2 + score3 + score4) / 4.0
WRITE (10,200) ave
WRITE (10,201) score1
WRITE (10,201) score2
WRITE (10,201) score3
WRITE (10,201) score4
:
```

Now repeat the problem above, but for 400 test scores.

Solution is **NOT** trivial!

Memory/Variables Problems

Any problem that involves accessing large sizes of data values in primary memory to perform calculations or output will lead to the same type of memory/variable accessing problems that cannot be practically solved using only simple variables.
Definitions:

collection of variables that all share the same name and the type; can be thought of as a list or table

Array : sequential list of variables of the same type.
Mathematics: \( a_1 \ a_2 \ a_3 \ \cdots \ a_i \ \cdots \ a_n \)
FORTRAN: \( a(1) \ a(2) \ a(3) \ \cdots \ a(i) \ \cdots \ a(n) \)

\[
\begin{array}{cccccccc}
(1) & (2) & (3) & \cdots & (i) & \cdots & (n) \\
\hline
a & & & & & & & \\
\end{array}
\]

Elements have same properties as simple variables of the same type.
Individual array elements are treated just like simple variables.
However, the entire array itself is NOT! The array name can appear by itself in certain cases!

INDEX SCORE access

\[
\begin{array}{c|c}
1 & 89 \ \text{score}(1) \\
\hline
2 & 78 \ \text{score}(2) \\
\hline
3 & 92 \ \text{score}(3) \\
\hline
4 & 84 \ \text{score}(4) \\
\hline
\vdots & \\
\end{array}
\]

the 1 is a subscript or index

The name of the whole array is SCORE.
We distinguish between individual elements of the array by giving a subscript: 1, 2, 3, etc.
Array Declaration

Declaration Examples

INTEGER, PARAMETER :: maxscores = 400, maxcredits = 5
INTEGER, PARAMETER :: maxstus = 5, numemps = 75
INTEGER :: score(maxscores), credits(maxcredits), idnum(maxstus)
REAL :: weight(maxstus), wages(numemps)
INTEGER :: arr(-2:4), b_arr(3,4)
INTEGER, DIMENSION(2) :: x_arr
INTEGER, DIMENSION(2:4) :: y_arr

Accessing Array Elements:
A subscript is used to access an array element.
Subscript must be an integer expression or constant

\[
\text{score}(1) = 0 \\
\text{score}(2) = \text{score}(1)
\]

\[
\text{DO } i = 1, \text{maxscores} \\
\quad \text{score}(i) = 0 \\
\text{END DO}
\]

\[
\text{sum} = 0 \\
\text{DO } i = 1, \text{maxscores} \\
\quad \text{sum} = \text{sum} + \text{score}(i) \\
\text{END DO}
\]

Remember: Array subscripts:
1) integer (expression)
2) must be in the proper range: lower bound to upper bound
Problem

Read 400 test scores (1 per line) and 1) find the average 2) print the average, 3) and finally print the 400 scores, (in original order).

Solution:

```fortran
PROGRAM example

IMPLICIT NONE
INTEGER :: count, total
INTEGER, PARAMETER :: maxscores = 400
INTEGER :: score(maxscores)
REAL :: average

total = 0
! input scores and sum
DO count = 1, maxscores
    READ(9,..) score(count)
    total = total + score(count)
END DO
! compute average & print
average = REAL(total) / REAL(maxscores)
WRITE(10,..) average
! output scores
DO count = 1, maxscores
    WRITE(10,..) score(count)
END DO
STOP
END PROGRAM example
```

score

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(2)</td>
<td>(3)</td>
<td>(400)</td>
</tr>
</tbody>
</table>
Array Input

Array Input Examples

Given the following array declarations:

```
INTEGER :: score(300), x(5), y(5), i, j
REAL :: charge(8)
```

**Array Input Code**

<table>
<thead>
<tr>
<th>Code</th>
<th>Resulting Actions</th>
</tr>
</thead>
</table>
| `DO i = 1, 300
  READ(9,..) score(i)` | reads 1 value from each of the next 300 lines |
| `DO j = 1, 5
  READ(9,..) x(j), y(j)` | reads 2 values from each of the next 5 lines |

**Automatic Array Input Examples (Short-List Techniques)**

Auto-Array I/O can only be used to input/output the entire array in sequence of the subscripts without other I/O, (of other variables or data) between the array elements.

```
READ(9,250) charge
250 FORMAT ( 8F6.2 )
```
reads 8 values from 1 line better to specify as ( 8(F6.2,TR1))

```
READ(9,260) charge
260 FORMAT (F6.2)
```
reads 8 values from 8 lines

```
READ(260) charge
260 FORMAT (2F6.2)
```
reads 8 values from 4 lines

```
READ(260) charge
260 FORMAT (4F6.2)
```
reads 8 values from 2 lines
Arrays 10.

### Previous Output Example

```fortran
DO count = 1, 400
    WRITE (10,300) score(count)
END DO
300 FORMAT (' ',I5)
```

Outputs 1 value per line, 400 lines.

### Implied DO-Loop

Abbreviated notation for Array I/O.

Specified on the same read/write line as the array.

Enclosed in parenthesis, with the array reference(s), omitting the DO.

Example:

```fortran
WRITE (10, 300) (score(count), count = 1, 400)
300 FORMAT (' ', I5)
```

Count serves as the LVC & must an integer variable.

The above implied DO outputs 1 value each on 400 lines.

If the following format statement had been used:

```fortran
300 FORMAT (' ', 4(I5,TR1))
```

The implied DO-loop would output 4 values per lines on 100 lines.

### Implied Input Loops

The following code segment shows an implied DO-loop with a read statement:

```fortran
INTEGER ::  day(15), idnum(15), i
READ (9, 200) (day(i), idnum(i), i = 1, 15)
200 FORMAT (I1, TR1, I3)
```

The implied DO-loop would input 2 values per line from the next 15 lines.
Standard Deviation

Std. Dev. Formula:

\[ \sigma = \sqrt{\frac{\sum_{i=1}^{N} (x_i - \mu)^2}{N}} \]

- \( \sigma \) is the standard deviation
- \( x_i \) represents data values
- \( N \) is the number of values
- \( \mu \) is the mean or average

Std. Dev. Program

```fortran
PROGRAM standev
  INTEGER :: i
  INTEGER, PARAMETER :: size=100
  REAL :: vals(size), sum, avg, difsum, stdev

  READ (9, ?) (vals(i), i= 1, size)  \input values

  sum = 0
  DO   i = 1, size  \find avg
    sum = sum + vals(i)
  END DO
  avg = sum / size

  difsum = 0  \determine sum of the squared differences
  Do   i = 1, size
    difsum = difsum + (vals(i) - avg)**2
  END DO

  stdev = SQRT( difsum/size)  \compute std dev
```

©1998 ND Barnette, DC Kar
An array can be declared with multiple dimensions.

Multiple dimensions get difficult to visualize graphically.
Two-Dimensional Arrays

Viewed as a table (rows, columns) of data

Given the following Population Data Table:

<table>
<thead>
<tr>
<th>columns</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blacksburg</td>
<td>23.0</td>
<td>24.0</td>
<td>24.5</td>
<td>28.0</td>
</tr>
<tr>
<td>C'Burg</td>
<td>8.0</td>
<td>9.0</td>
<td>7.0</td>
<td>8.5</td>
</tr>
<tr>
<td>Radford</td>
<td>4.0</td>
<td>4.5</td>
<td>5.2</td>
<td>5.6</td>
</tr>
<tr>
<td>Pulaski</td>
<td>18.1</td>
<td>19.2</td>
<td>20.1</td>
<td>20.2</td>
</tr>
<tr>
<td>Dublin</td>
<td>8.6</td>
<td>7.6</td>
<td>6.0</td>
<td>2.1</td>
</tr>
</tbody>
</table>

Consider this info to be stored in an array POP declared as:

```fortran
real POP(5,4)
```

What is the population of ..

Radford in 1991?
4.5 thousand = POP(3,2)

Christiansburg in 1992?
7.0 thousand = POP(2,3)
2-D Array

Nested DO-Loops

2-D arrays are processed using a DO-loop with an inner DO-loop
Given the declaration:

```
REAL :: pop(5,4)
```

Examples:

Print out the array, one number per line row by row:
```
DO  r = 1, 5
    DO c = 1, 4
        WRITE(10,..) pop(r,c)
    END DO
END DO
```

Read one number per line, column by column
```
DO  c = 1, 4
    DO r = 1, 5
        READ(9,..) pop(r,c)
    END DO
END DO
```

READ will be executed as ..
```
pop(1,1)
pop(2,1)
pop(3,1)
pop(4,1)
pop(5,1)
pop(1,2)
pop(2,2)
etc.
```
Examples (cont)

What if each line of the input data has a row of the data?
How can we read it into the array properly?

\[
\begin{align*}
\text{DO } r &= 1, 5 \\
\text{read}(9,..) &\text{(pop(r,c), c = 1, 4)} \\
\text{END DO}
\end{align*}
\]

What if all the data is on one line?

\[
\begin{align*}
\text{READ}(9,..) &\text{ pop} \quad \text{Short-List method (Do NOT do it this way).} \\
\text{READ}(9,..) &\text{ ((pop(r,c), c = 1, 4), r = 1, 5)} \\
\text{read all columns for each row} \\
\text{read all rows}
\end{align*}
\]
Problem: Write a program to read in the city population data file and compute the population averages for each year, for every city and for all of the cities for all of the years.

For each year determine the average population for all of the area cities.
For each city determine the average population during the given years.
Determine the average city population for all of the given years.

<table>
<thead>
<tr>
<th>columns</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blacksburg</td>
<td>23.0</td>
<td>24.0</td>
<td>24.5</td>
<td>28.0</td>
<td></td>
</tr>
<tr>
<td>C'Burg</td>
<td>8.0</td>
<td>9.0</td>
<td>7.0</td>
<td>8.5</td>
<td></td>
</tr>
<tr>
<td>Radford</td>
<td>4.0</td>
<td>4.5</td>
<td>5.2</td>
<td>5.6</td>
<td></td>
</tr>
<tr>
<td>Pulaski</td>
<td>18.1</td>
<td>19.2</td>
<td>20.1</td>
<td>20.2</td>
<td></td>
</tr>
<tr>
<td>Dublin</td>
<td>8.6</td>
<td>7.6</td>
<td>6.0</td>
<td>2.1</td>
<td></td>
</tr>
<tr>
<td>averages</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>overall average</td>
</tr>
</tbody>
</table>

rows: thousands

Declare an "extra" row & column in the table to store the averages.
PROGRAM POPULOUS
IMPLICIT NONE
INTEGER :: row, col
INTEGER, PARAMETER :: cities = 6, years = 5
REAL :: popdata(cities,years), yearsum(years)
REAL :: yearcitysum, citysum

OPEN ( 9, FILE = 'CITYPOP.DAT')
OPEN (10, FILE = 'POPAVG.RST' )

DO row=1, cities-1
   READ (9, 100) (popdata(row,col) , col=1, years-1)
END DO
dyacitysum = 0.0
DO yr = 1, years-1
   yearsum(yr) = 0.0
END DO
DO row=1, cities-1
   citysum = 0.0
   DO col= 1, years-1
      citysum = citysum + popdata(row,col)
      yearsum(col) = yearsum(col) + popdata(row,col)
   END DO
   yearcitysum = yearcitysum + citysum
END DO
popdata(row, years) = citysum / REAL(years-1)
END DO
DO yr = 1, years-1
   popdata(cities, yr)= yearsum(yr) / REAL(cities-1)
END DO
popdata(cities, years)= yearcitysum / REAL((cities- &
1)*(years-1))

DO row=1, cities
   WRITE (10, 700) (popdata(row,col) , col=1, years)
END DO
CLOSE(9)
CLOSE(10)
STOP

100 FORMAT ( 4(F5.1, TR1) )
700 FORMAT ( 5(F5.1, TR1) )

END PROGRAM POPULOUS
Population Table Program

Typing in the input given in the table on slide 9.12:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>23.0</td>
<td>24.0</td>
<td>24.5</td>
<td>28.0</td>
<td></td>
</tr>
<tr>
<td>8.0</td>
<td>9.0</td>
<td>7.0</td>
<td>8.5</td>
<td></td>
</tr>
<tr>
<td>4.0</td>
<td>4.5</td>
<td>5.2</td>
<td>5.6</td>
<td></td>
</tr>
<tr>
<td>18.1</td>
<td>19.2</td>
<td>20.1</td>
<td>20.2</td>
<td></td>
</tr>
<tr>
<td>8.6</td>
<td>7.6</td>
<td>6.0</td>
<td>2.1</td>
<td></td>
</tr>
</tbody>
</table>

The given program produces the (woefully unlabelled) output:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>23.0</td>
<td>24.0</td>
<td>24.5</td>
<td>28.0</td>
<td>24.9</td>
<td></td>
</tr>
<tr>
<td>8.0</td>
<td>9.0</td>
<td>7.0</td>
<td>8.5</td>
<td>8.1</td>
<td></td>
</tr>
<tr>
<td>4.0</td>
<td>4.5</td>
<td>5.2</td>
<td>5.6</td>
<td>4.8</td>
<td></td>
</tr>
<tr>
<td>18.1</td>
<td>19.2</td>
<td>20.1</td>
<td>20.2</td>
<td>19.4</td>
<td></td>
</tr>
<tr>
<td>8.6</td>
<td>7.6</td>
<td>6.0</td>
<td>2.1</td>
<td>6.1</td>
<td></td>
</tr>
<tr>
<td>12.3</td>
<td>12.9</td>
<td>12.6</td>
<td>12.9</td>
<td>12.7</td>
<td></td>
</tr>
</tbody>
</table>
EXAMPLES:

```
INTEGER, DIMENSION(10):: arr = (/1,0,9,5,6,4,2,3,8,2/)
INTEGER:: i
INTEGER, DIMENSION(5):: b_arr = (/ (0, i = 1, 5) /)
INTEGER, DIMENSION(10) :: c_arr = (/0,0, (i, i = 3, 7), 0, 5,3/ )
```

```
INTEGER:: x_arr(20), y_arr(30)
x_arr=(/ (0, i = 1, 10)/ )
y_arr=(/(2*i, i = 1, 30, 2)/)
y_arr=(/(i, i = 2, 30, 2)/)
```

AGGREGATE OPERATIONS ON ARRAYS

Fortran 90 language supports aggregate (element-by-element) operations on arrays provided that the arrays and scalars involving in an operation are conformable to each other and the operation is defined between two conformable objects.

EXAMPLES:

```
! The following declaration will declare and set all elements of arrays x,!
y, and z to zeros.
REAL, DIMENSION(20)::x=0.0, y=0.0, z=0.0
REAL :: array_max

! The following statement will multiply 10 to every element of x.
x = 10 * x

! The following statement will multiply corresponding elements of x and!
y and set the corresponding elements of z
z = x * y

! The following statement will assign the maximum of x, y, and z to!
array_max by using the intrinsic function MAX.
array_max = MAX (x, y, z)
```