Chapter 8

Arrays

All variables we have used so far are scalar single values, meaning that the storage space is reserved to store only real or one double precision. Often time however we need to work with a collection of these variables. Many engineering and scientific calculations also involve vectors and matrices which are rectangular arrays of numbers. Fortran provides a special data structure called an array to represent vectors and matrices. An array is essentially an ordered list of scalar elements, each of which can be referred to by its location in the array.

8.1 Vectors

The program in 8.1 declares two arrays, sets their value, and compute the mean of one of them. The new feature introduced here is in line 4 of the program where two vectors are declared, x(11) and f(11) whose dimension is 11; this is done simply by appending the size in parenthesis, (11), to the name of the array. There are various ways of declaring arrays in FORTRAN, depending on where in the code they are declared (i.e. a main program, a subroutine or in a module). However, array declarations with known dimensions at compile time are allowed anywhere in the code, i.e. the dimensions should be either constants (such as the number 11 in the example) or named constants declared in an integer, parameter statement. The latter is preferred in large codes as it simplifies changing the array dimension later by specifically naming the array dimension. The following are valid array declarations in classical FORTRAN

```fortran
integer, parameter :: n=200
real, dimension(n) :: x,f
real :: y
dimension y(n)
```
By default the index of a FORTRAN array starts at 1 (it starts at 0 in C). The default can be overridden by specifying explicitly the bounds of an array as in
\[
\text{integer :: xcord(-2:20)}
\]
In this case the array index starts at -2 and ends at 20 so that the array has \((20 + 2 + 1) = 23\) elements.

In classical FORTRAN the elements of an array are stored in contiguous memory locations, so that \(x\) occupies a total of 11 adjacent words. An element outside the dimension of an array, such as \(x(0)\) or \(x(12)\), is a memory location that is probably occupied by some other data such as other variables in the program or by some machine instruction belonging the present executable or possibly the operating system. Unpredictable behavior ensues if your code tries to use an array element outside the dimensioned size of the array. The particular behavior depends on the data that happens to be in that memory location. In the actually happy event that you try to access operating system memory a segmentation fault occurs; the program dies an ignominious death and you know for sure something in the code is wrong. In other circumstances the erroneous value used does not cause the code to crash but produces wrong or unexpected results but without any warning that something went wrong. Such out of bounds array indexing must be zealously guarded against by carefully checking the array indexing logic in the code: never refer to an array index outside the dimensioned size. It is a very good idea to turn array bounds checking in the compiler options when you are testing your code. This is usually \(-C\) in most compilers: it causes the compiler to issue a detailed warning of where the bound trespass occurred. Array bounds checking slows execution and should be turned off once you are ready for production runs.

### 8.2 Matrices, and Multidimensional Arrays

Matrices in fortran are represented by arrays with two dimensions. For example a two and three dimensional array would be declared as follows

\[
\text{integer :: i2d(4,3), i3d(4,5,6)}
\]

The matrix \(i2d\) has 4 rows (the left most dimension) and 6 columns (the right most dimension), and is easy to visualize as shown below:

\[
\begin{array}{ccc}
\text{i2d(1,1)} & \text{i2d(1,2)} & \text{i2d(1,3)} \\
\text{i2d(2,1)} & \text{i2d(2,2)} & \text{i2d(2,3)} \\
\text{i2d(3,1)} & \text{i2d(3,2)} & \text{i2d(3,3)} \\
\text{i2d(4,1)} & \text{i2d(4,2)} & \text{i2d(4,3)} \\
\end{array}
\]

It is of course harder to visualize three-dimensional arrays unless one uses a “depth” for the third, right most, dimension; one can then imagine 6 two dimensional slices of \(i3d(4,5,:)\). More array dimension can be declared simply by making the parenthesized list of dimensions longer. Standard fortran supports up to 7
dimensional arrays. The way the arrays are arranged in memory is an important topic that has bearing on how to pass arrays to subroutines. But before we get to that it is informative to present some jargon associated with arrays:

- **rank** is the number of dimensions of an array. A vector has rank 1, \( i2d \) has rank 2 and \( i3d \) has rank 3.

- **extent** is the number of elements in a dimension. For example \( i3d \) has an extent of 4 along dimension 1, 5 along dimension 2, and 6 along dimension 3.

- **size** is the total number of elements in an array and is simply equal to the product of the extents. \( i2d \) has size 12 and \( i3d \) has size 120.

- **shape** is an ordered list of the extents so that \( i3d \) has shape (4,5,6).

Do loops and arrays go hand in hand. Accessing the members of a multi-dimensional array requires nested do loops:

```
   do j = 1,3
      do i = 1,4
         print *,i2d(i,j)
      enddo
   enddo
```

```
   do i = 1,4
      do j = 1,2
         print *,i2d(i,j)
      enddo
   enddo
```

The code on the left access the array column-wise whereas the code on the right accesses it row-wise. The index notation allows easy access to all the array members in any order, however, access by column-wise order is the most efficient since it takes advantage of of memory traffic rules in modern computers. For arrays with large dimensions the left hand code would execute substantially faster then the right hand code.

### 8.3 Layout of arrays in memory

Computer memory is strictly linear with one byte following another like the box-cars in a railroad train. So there is no natural way of storing 2-dimensional, let alone multi-dimensional data. So multi-dimensional arrays must be **folded** into one-dimensional sequences of memory locations, i.e. the visual is a line rather then a rectangle. FORTRAN stores arrays in **column-major order**: the first column is listed first from top to bottom, followed by the second column, etc. Thus the left most index is changing fastest unlike an odometer for example. The matrix

\[
M = \begin{bmatrix}
11 & 12 & 13 & 14 \\
21 & 22 & 23 & 24 \\
31 & 32 & 33 & 34 \\
\end{bmatrix}
\]
would be arranged as follows in memory:

\[
\begin{array}{cccc}
11 & 21 & 31 & 12 \\
M_{11} & M_{21} & M_{31} & M_{12} \\
22 & 32 & 13 & 23 \\
M_{22} & M_{32} & M_{13} & M_{23} \\
33 & 14 & 24 & 34 \\
M_{33} & M_{14} & M_{24} & M_{34}
\end{array}
\]

It is useful to relate the indices \((i, j)\) of an element entry to a single-index reference \(l\) which represents the location in memory after the element of the matrix. For a 2D array declared as \(A(M,N)\) it is

\[
l = (j - 1)M + j
\]

Likewise the single index for a 3D array declared as \(B(M,N,P)\) the single index of the triplet \((i, j, k)\) is

\[
l = (k - 1)MN + (j - 1)N + i
\]

The inverse transformation is sometimes useful, going from a single index to a multi-index notation. For a 2 and 3D array they are:

\[
\begin{align*}
\left\{ \begin{array}{c}
j = ((l - 1)/M) + 1 \\
i = l - (j - 1)M
\end{array} \right. & \quad \text{and} \quad \left\{ \begin{array}{c}
k = (((l - 1)/(MN)) + 1 \\
j = ((l - (k - 1)MN))/M + 1 \\
i = l - (l - 1)MN - (j - 1)M
\end{array} \right.
\end{align*}
\]

where we have assumed integer arithmetic in all operations.

8.3.1 Efficient array access

We are now in a position to explain the efficiency of accessing memory by column-wise order. If an array cannot fit in the fast memory inside a CPU portions of it must reside in the computer’s main memory where access time is slow; access to the entire array would then require shuttling the data piecewise to the CPU registers where they are acted upon. The CPU brings contiguous arrays chunks into memory to save time instead of element by element, since every time the CPU reaches into main memory a time-penalty is exacted. Accessing the array in its contiguous order the way it is laid out in memory ensures that this penalty is minimized. If the array fits in registers then these considerations do not apply. Recently computer manufacturers have been adding different of types of memory to computers, so-called L1 and L2 cache memory to mitigate the slowness of main memory; the term non-uniform memory access (NUMA) refers to the implementation of different memory hierarchies with different access speeds.

8.4 Passing arrays to subroutines

The passing of array arguments, particularly multi-dimensional arrays, to subroutine is somewhat of a delicate issue that relates directly to the layout of arrays
in memory. When an array is passed to a subroutine what is actually passed is the memory address of the first array element. So for a subroutine to manage the array elements correctly it has to be informed about the rank, and the extent of the dimensions. A subroutine that transposes a matrix would have the following code:

```fortran
program array2D
    integer, parameter :: m=3, n=4
    integer :: A(m,n), AT(n,m), B(m*n), BT(n,m), C(m+1,n), CT(n,m)
    integer :: i, j, ij
    ij = 0
    do j = 1, n
        do i = 1, m
            ij = ij + 1
            A(i,j) = (n-1)*(j-1) + i; B(ij) = A(i,j); C(i,j) = A(i,j)
        enddo
        C(m+1,j) = 0
    enddo
    call Transpose1(A, AT, m, n) ! OK since the entire array is passed
    call PrettyPrint(A, m, m, n)
    call Transpose2(B, BT, m, n, m+1, n) ! OK since leading dimension of C is passed.
    call PrettyPrint(B, m, m, n)
    call PrettyPrint(CT, n, n, m)
    call PrettyPrint(CT, n, n, m)
    stop
contains
    subroutine transpose1(P, Q, m, n)
        implicit none
        integer, intent(in) :: m, n
        integer, intent(in) :: P(m,n)
        integer, intent(out) :: Q(n,m)
    end subroutine transpose1
end program array2D
```
The first call to transpose would accomplish the desired result since the entire matrices are passed to the subroutine. Sometimes it is useful to declare a large vector in the program and use it as a matrix inside the subroutine. This is possible if care is taken in declaring enough element for the vector.
8.5 Allocatable Arrays

The new FORTRAN standard (90 and newer) allows the dynamic allocation of memory via the allocate statements. FORTRAN is a little strict about how it allocates arrays, particularly as it seeks to preserve the optimization of memory access. An array has to be declared explicitly allocatable before one can allocate memory for it at run time. This is accomplished with the allocatable attribute. The following example allocates a two-dimensional array:

```fortran
program aralloc
  implicit none
  integer :: m,n, i,j
  integer, allocatable :: ind(:,:)
  
do
    read *, m,n
    if (m*n .ne. 0 .and. n>0) then
      if(allocated(ind)) deallocate(ind) ! deallocate before reallocating
      allocate(ind(m,n)) ! allocate array
    else
      exit
    endif
  do j = 1,n
    do i = 1,m
      ind(i,j) = (j-1)*m+i
    enddo
  enddo
  do i = 1,m
    do j = 1,n
      write(6,fmt='(i3,$)') ind(i,j)
    enddo
  enddo
  write(6,*)
  enddo
  end program aralloc
```

The example also includes a number of functions that can be used to inquire the status of allocatable objects. Allocatable arrays in FORTRAN must have their rank declared, you cannot allocate a two-dimensional array declared as a one-dimensional array. The restriction is meant to allow the compiler to carry out the same type of optimization performed on allocatable arrays as regular arrays.

- The allocatable attribute indicates that memory for the array will be de-
cided at run-time, and the compiler should defer the memory allocation. The rank of the array is declared by the comma separated list of colons. The extent along each dimension and the size are deferred until execution time.

- The allocate statement makes available the requested storage and returns the address of the first element. The allocate statement accepts an option argument that lets the user know whether the memory allocation was successful.

\[
\text{allocate}\left( \text{ind(m,n)}, \text{stat} = \text{is} \right)
\]

The stat keyword refers to the name of the optional argument whose value is returned in integer is; a non zero value indicates an error.

- It is an error to write/read from an un-allocated array

- The deallocate statement returns the memory to the Operating System. The array cannot be addressed and the values stored in it are lost.

- The allocated function returns a logical .true. if the array is allocated and a .false. otherwise.

### 8.6 Array Sections

FORTRAN 90 allows the manipulation of portions of the array by using the colon notation. For example the following code assign the value 1 to the first three elements of the array and the value two to the rest.

```fortran
program arsec
  implicit none
  integer :: id(6), ja(4,2), ka=(5,5)
  id(1:3) = 1; id(3:6)=2
  id(1:5:2) = 1  ! odd index elements id(1), id(3), id(5)
  id(2:6:2) = 6  ! even index elements id(2), id(4), id(6)
  ja = ka(1:4,1:2)
  stop
end program arsec
```

The information triplets represent a sort of implied do-loop and has the form

\[
\text{expression1} : \text{expression2} : \text{expression3}
\]
Each of the expressions must evaluate to an integer, the first expression gives a lower bound, the second an upper bound, and the third the stride between the elements. If the lower bound is omitted it defaults to 1, if the upper bound is omitted it defaults to the maximum bound allocated or declared, and finally if the third expression is omitted it defaults to 1.

The sections can be used effectively in multi-dimensional array even though the margin of error increases with the array rank. The following example copies a $4 \times 2$ section of a $5 \times 5$. The most common useful of this construct is to pass rows of the matrix to subroutines or functions:

```fortran
do i = 1,nrows
   a = ddot(ra(i,1:ncols), vec, ncols)
enddo
```

However this practice is discouraged as it involves copying the data to a temporary array at the entrance and exit to the subroutine; the penalty in CPU time increases with the size of the array.