Introduction to
Scientific & Engineering Computing
BIL 102FE (Fortran) Course
for
Week 7

Dr. Ali Can Takinacı

Assistant Professor

in

The Faculty of Naval Architecture and Ocean Engineering

80626

Maslak – Istanbul – Turkey

AN INTRODUCTION TO ARRAYS

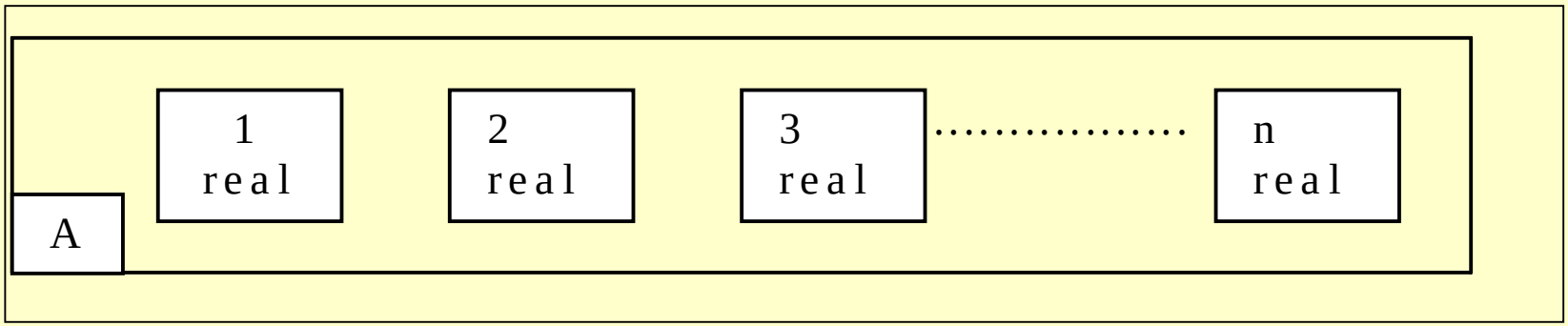
In scientific and engineering computing, it is very common to need to manipulate ordered sets of values, such as vectors and matrices.

There is also a common requirement in many applications to repeat the same sequence of operations on successive sets of data.

This chapter explains the principles of array processing features and for introductory purposes but they are restricted to one-dimensional arrays

The array concept

An **array** is identified by the same name but with an index, or **subscript** to identify individual locations can repeat the statements in a loop or use different variables for each iteration to carry out the same operation.



The whole set of n boxes is called A, but within the set it can be identified individual boxes by their position within the complete set. Mathematicians are familiar with this concept and refer to an ordered set like this as the *vector* A and the individual elements as A_1, A_2, A_3 .... A_N.

This ordered set of related variables are called as an **array** and the individual items within the array as **array elements**

A(1), A(2), A(3), ..., A(n)

An array element is defined by writing the name of the array followed by a subscript, where the subscript consists of an integer expression (known **as the subscript expression**) enclosed in parentheses.

Thus, if x, y and z are arrays, of any type, and i,j, k are integer variables, then the following are all valid ways of writing an array element:

```
x(10)
y(i+4)
z(3*max(i,j,k))
  x(int(y(i)*z(j)+x(k)))
```

Array element is a scalar object and may be used as such.

```
print *, x(5)
x(1) = x(2) + x(4) + 1
```

Array declaration

When an array is declared, however, the compiler will allocate several storage units, which can be done by using **dimension attribute**.

For example the declaration of three real arrays each containing 50 elements will be:

```
real, dimension(50) :: a, b, c
```

This informs the compiler that each of three variables specified is an array having 50 elements. The arrays having different sizes must be written separately.

```
real, dimension(50) :: a, b, c
real, dimension(20) :: x, y
```

By default, the subscripts will start at 1; if one wishes the subscript to have a different range of values then it may be written this by providing the lower bound and the upper bound explicitly separated by a colon:

```
real, dimension(11:60) :: a, b, c
real, dimension(-20:1) :: x
real, dimension(0:49) :: z
```

- F permits up to seven subscripts, each of which relates to one **dimension** of the array. For each dimension there are two bounds which define the range of values that are permitted for the corresponding subscript: the lower and the upper bound.

- The number of permissible subscripts for a particular array is called its **rank**.

- The extent of a dimension is the number of elements in that dimension and is equal to the difference between the upper and lower bounds for that dimension plus one.

- The size of an array is the total number of elements, which make up the array.

- The shape of an array is determined by its rank and the extent of each dimension.

Array constants and initial values

Since an array consists of a number of array elements, it is necessary to provide values for each of these elements by means of an **array constructor**. It consists of a list of values enclosed between special delimiters:

```
arr = (/ value_1, value_2, ... /)
```

If arr is an integer of size 10, its elements could be set to the values 1, 2, ..., 10 by the statement:

```
arr = (/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/)
```

Since the use of this representation for high numbered arrays is impractical, in general**, an implied do** is used. It takes the general form

`(`*`value_list, implied_do_control`*`)`

where the *`implied_do_control`* takes exactly the same structure as the **do** variable control specification in **a do** statement. Thus, the assignment statement shown above for the array `arr` could also be written in more compact and less error-prone form.

```
arr = (/(i, i = 1 , 10)/)
```

The index in an implied do (here i) must not be used for any other purpose except as an implied do control in another **array constructor** in the same procedure or main program. Thus, for example, i could not also be a dummy argument in the procedure that contained the array constructor.

```
arr = (/ -1, (0, i = 2 , 49), 1/)
```

This array constructor defines the sequence of 50 values which are all zero except for the first, which takes the value –1, and the last, which takes the value 1.

Input and output with arrays

The following statement would output the five elements of the array p followed by the third and fourth elements of the array q and the whole of the array r:

```
real, dimension(5) :: p, q
integer, dimension(4) :: r
print *, p, q(3), q(4), r
```

In a similar way

```
integer, dimension(5) :: value
read *, value
```

would read five values from the input stream.

Using arrays and array elements in expressions and assignments

An array element can be used anywhere, that a scalar variable can be used. It identifies a unique location in the memory to which a value can be assigned or input and whose value may be used in an expression or output list. The great advantage is that, by altering the value of the array element's subscript.

The use of array variables within a loop, therefore, greatly increases the power and flexibility of a program.

The rules for working with whole arrays:

- Two arrays are conformable if they have the same shape.
- A scalar, including a constant, is conformable with an array.
- All intrinsic operations are defined between two conformable objects.

```
      .
      .
    real, dimension(20) :: a, b, c, d
      .
      .
!   whole-array style manipulation
    a = c*d
      .
!   element-by-element style array manipulation
    do i = 1 , 20
         b(i) = c(i) * d(i)
    end do
      .
      .
```

It is obvious that the first style (whole-array style manipulation) is much easier to read than the second is, as well as avoiding the need for extraneous do loop variable i

An important point to notice is that the rule is that the *shapes* of two arrays must be the same for them to conformable. It does *not* mean that the range of the subscripts need be the same.

```
        .
      real:: a(1:20), b(0:19), c(10:29), d(-9:10)
       .
       .
!     whole-array style manipulation
      a = c*d
       .
!   element-by-element style array manipulation
    do i = 1 , 20
         b(i-1) = c(i+9) * d(i-10)
    end do
      .
```

This program fragment is exactly the same as that above, except that the bounds of four arrays are all different, even though their extents are the same.

A scalar is conformable with any array means that it can be written statements such as

```
array_1 = 10*array_2
```

which will cause every element of the array `array_1`, whatever its shape, to be assigned a value 10 times the corresponding element of the array `array_2,` as long as its shape is the same as that of `array_1.` Furthermore, it means that

```
real, dimension(1000) :: arr
arr = 1.0
```

will set every element of the array `arr` to one. This would work regardless of the rank or size of `arr.` It also means that all the elements of an array in a module or a procedure may be initialised in exactly the same way as for scalar variables:

```
real, save :: a = 0.0, b = 0.0
real, dimension(50), save :: c = 0.0, d = 10.0
```

sets all 50 elements of c to the value zero and all-50 elements of d the value 10. Arrays, like scalars, may not be initialised in a main program.

Using intrinsic procedures with arrays

In F, arrays may be used as arguments to many of the intrinsic procedures in just the same way scalars are. If an elemental function has an array as an argument then the result of the function will be an array with the same shape as the argument. Thus, if `array_1` and `array_2` are conformable arrays, the statement,

```
array_1 = sin(array_2)
```

assigns the sine of each element of the array `array_2` to each corresponding element of the array `array_1.` Where an intrinsic function has more than one argument then they must all be conformable, as would be expect. Thus the statement

```
arr_max = max(100.0, a, b, c, d, e)
```

will assign to the array elements of `arr_max` the maximum value of the corresponding elements of the arrays `a, b, c, d` and `e` or 100.0 if that is greater as long as the six arrays `arr_max, a, b, c, d` and `e` are all conformable; the scalar value 100.0 is conformable with any array.

There are a number of intrinsic functions especially meant for dealing with arrays, for example if `arr` is a rank-one array

- `maxval(arr)` The maximum value of the elements of `arr`
- `maxloc(arr)` The location of the first element of `arr` having the value `maxval(arr)`
- `minval(arr)` The minimum value of the element `arr`
- `minloc(arr)` The location of the first element of `arr` having the value `minval(arr)`
- `size(arr)` The number of elements in `arr`
- `sum(arr)` The sum of the elements of `arr`

Sub-arrays

A sub-array consists of a selection of elements of an array and can be manipulated in the same way that a whole array can.

In F, **array sections** can be extracted from a parent array in a rectangular grid using **subscript triplet** notation, or in a completely general manner using **vector subscript** notation. A subscript triplet takes the following form:

```
subscript_1 : subscript_2 : stride
```

Thus, if the array `arr` is declared as

```
real, dimension(10), arr
```

2:8:3 is a subscript triplet that defines a set of integers that starts at 2 and proceeds in increments of 3 until 8 is reached. Consequently, the set of subscripts is 2, 5 and 8, and `arr`(2:8:3) is an array whose elements are `arr`(2), `arr`(5) and `arr`(8) in that order.

If stride is negative, then the subscript order is reversed with the result that `arr`(8:2:-3) is an *array* whose elements are `arr`(8), `arr`(5) and `arr`(2), in that order.

`arr(1:10)` is a rank-one real array containing all the elements of arr; it is in fact, identical to `arr`

`arr(3:5)` is a rank-one real array containing the elements `arr`(3), `arr`(4) and `arr`(5)

`arr(:9)` is a rank-one real array containing the elements `arr`(1), `arr`(2), ...,`arr`(9)

`arr(::4)` is a rank-one real array containing the elements `arr`(1), `arr`(5), and `arr`(9)

`arr(:)` is a rank-one real array containing all the elements `arr;` it is the same as `arr`

A simple example of how array sections can simplify code can be seen if it is considered how to print the first three elements of an array work of size `n`. Rather than write:

```
print *, work(1), work(2), work(3)
```

it can be written the simpler

```
print *, work(1:3)
```

As more complicated example, if the even-numbered elements of an array is wanted to print. Rather than write:

```
integer :: i
do i = 2, n, 2
print *, work(i)
end do
```

or the simpler form

```
print *, work(2 :: 2)
```

This defines an array section consisting of the even-numbered elements of `work`. This version is both easier to read and less error-prone than the first method.

```
      integer, dimension(10) :: a
      integer, dimension(3) :: b
      .
      .
      .
!     set b(1)=a(4), b(2)=a(5) and b(3)=a(6)
      b = a(4:6)
!     set a(1), a(2), a(3) to 0
      a(1:3) = 0
!     set the odd-numbered elements of a to 1
      a(1 :: 2) = 1
!     make each odd-numbered element of a equal to
!     one more than the next even-numbered element
```

Array sections, since they are arrays, can be used in conjunction with the intrinsic procedures of F. Thus:

```
real, dimension(4) :: a
 .
 .
 .
print *, sum(a(2:4))
print *, sum(a(1:4:2))
```

prints out the value of a(2)+a(3)+a(4) and then the value of a(1)+a(3).

A vector subscript can be used to construct a vector from an array that is longer than that array; for example if the arrays p and u are declared as follows:

```
logical, dimension(3) :: p
integer, dimension(5) :: u = (/3, 2, 2, 3, 1/)
```

then p(u) is a rank-one logical array of size 5 whose elements are, in order, p(3), p(2), p(2), p(3) and p(1). This is an example of **a many-one array section**. That is, it is an array section with a vector subscript having at least two elements with the same value.

Arrays and procedures

In F, one of the most important aspects of the argument-passing mechanism is that a procedure does not need to know the details of the calling program unit and the current procedure except the information about its arguments.

It would make no sense at all for the bounds of a dummy argument array to be fixed and for all arrays passed as actual arguments to be required to have the same bounds.

Using the assumed-shape array does this. If an array is a procedure dummy argument, it *must* be declared in the procedure as an assumed-shape array.

An assumed-shape array is a dummy argument array whose shape is not known but which *assumes* the same shape as that of any actual argument that becomes associated with it.

An assumed-shape array can take one of two forms:

`(`*`lower_bound`*`:)`

or, simply

`(:)`

The second for is equivalent to the first with a lower bound equal to 1.

In both cases the upper bound will only be established on entry to the procedure. Therefore, the actual argument array overlays the dummy argument array.

```
program main
      .
      .
    real, dimension(4) :: a
      .
      .
    call sub(a)
      .
      .
    end program main
    subroutine sub(x)
    real, intent(in), dimension(:) :: x
      .
```

Here x(1) corresponds to a(1)

x(1) => a(1)

x(2) => a(2)

x(3) => a(3)

x(4) => a(4)

A more complicated case:

```
program main
          .
real, dimension(b:40) :: a
  .
call sub(a)
  .
end program main
subroutine sub(x)
  .
real, intent(in), dimension(d:) :: x
  .
```

from declarations x(d) will correspond to a(b)

x(d) => a(b)
x(d+1) => a(b+1)
x(d+2) => a(b+2)
x(d+3) => a(b+3)
.
.

Another example will make this concept clearer

```
 program main
  .
real, dimension(10:30) :: a, b
  .
call sub(a,b)
  .
  .
end program main
subroutine sub(dummy_array_1, dummy_array_2)
  .
real, intent(inout), dimension(:) :: dummy_array_1,
                                     dummy_array_2
              .
```

The two dummy argument arrays will both have lower bounds of 1 and upper bounds of 21

If a subroutine is subsequently called from another program unit that contains the declarations

```
real :: p(-5:5), q(100)
```

by the statement

```
call sub(p, q)
```

On this occasion both dummy argument arrays will have a lower bound of 1, while the upper bound of dummy_array_1 will be 11 and the upper bound of dummy_array_2 will be 100.

For many purposes all the use of array manipulations examined above may be needed, but there also be occasions when it will be necessary for a procedure to know the size of the actual arrays associated with its dummy argument arrays.

The intrinsic function, **`size`**, can solve this problem.

Thus, for rank-one array argument `arr`, **`size`**`(arr)` returns the number of elements in `arr`.

Example: Write a subroutine that will sort a set of 10 numbers in ascending (from smaller to bigger) order.

Analysis: This is quite a simple method to code in F; The structure plan:

subroutine sort_ascending(*name*)

1. repeat for *i* from 1 to *number*-1

1.1 Save *i* and *name(i)* as current “earliest name”

1.2 Repeat for *j* from *i*+1 to *number*

1.2.1 if *name(j)* is “earlier” than current ‘earliest’ store it and its index

1.3 If step 1.2 found an “earlier” name with *name(i)*

```
      subroutine sort_ascending(number,name)
!     A subroutine to sort a set of integer numbers into ascending order
!     dummy argument
      integer, dimension(*), intent(inout) :: name
      integer, intent(inout) :: number
!     local variables
      integer :: save_index, i, j, first, temp
!     loop to sort number-1 into order
      do i = 1 , number-1
!     initialize earliest so far to be the first in this pass
      first = name(i)
      save_index = i
!     search remaining (unsorted items) for earliest one
      do j = i+1 , number
      if (name(j) < first) then
      first = name(j)
      save_index = j
      end if
      end do
      if (save_index /= i) then
      temp = name(i)
      name(i) = name(save_index)
      name(save_index) = temp
      end if
      end do
      end subroutine sort_ascending
```