# 5 Arrays and Formatted I/O

## 5.1 Aims

By the end of this worksheet you will be able to:

- ❑ Understand the use of arrays
- ❑ Improve the appearance of your output

## 5.2 Arrays

Let us imagine that we want to find the average of 10 numbers. One (crude) method is shown in the next program.

```
program av
real :: x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,average
read *, x1,x2,x3,x4,x5,x6,x7,x8,x9,x10
average= (x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10)/10
print *, 'the average is ',average
print *, 'the numbers are:'
print *, x1
print *, x2
print *, x3
print *, x4
print *, x5
print *, x6
print *, x7
print *, x8
print *, x9
print *, x10
end program av
```

This approach is messy, involves a lot of typing and is prone to error. Imagine if we had to deal with thousands of numbers!

The way around this is to use **arrays**. An array is a list that we can access through a subscript. To indicate to FORTRAN that we are using an array, we just specify its size when we declare it.

```
real, dimension(100)  ::x
    .
    .
x(1) = 3
x(66) = 4
```

This snippet of code allocates 100 memory locations to the array **x**. To access an individual location, called an **array element**, we use a **subscript** – here we are assigning the number 4 to the 66$^{th}$ element of array x and 3 to the 1$^{st}$ element.

Now let's return to program **av** at the start of this worksheet, we'll re-write it using an array.

```
program av2
implicit none
real ,dimension(10) :: x
real                :: average,sum
integer             :: i
print *, 'enter 10 numbers'
sum=0.0
do i=1,10
     read *, x(i)
     sum=sum+x(i)
end do
average=sum/10
print *, 'the average is ',average
print *, 'the numbers are'
print *,x
end program av2
```

Notice that if we type

```
print*, x
```

the program will print out the entire contents of the array.

The additional benefit of this program is that with very few changes, we could make it deal with any number of items in our list. We can improve on this still further by making use the **parameter** data type:

```
        program av3
!just change the value of the parameter to change the size of the
!array
        implicit none
        integer, parameter    :: imax = 10
        real,dimension(imax)  :: x
        real                  :: average,sum
        integer               :: i
        print *, 'enter' ,imax, ' numbers'
        sum=0.0
        do i=1,imax
             read *, x(i)
             sum=sum+x(i)
        end do
        average=sum/imax
        print *, 'the average is ',average
        print *, 'the numbers are'
        print *,x
        end program av3
```

**Note** this is an example of good programming. The code is easily maintainable – all we have to do to find an average of a list of numbers of *any* size is just to change the size of the parameter **imax**. We can also allocate the size of the array at **run time** by dynamically allocating memory.

The following program demonstrates the use of arrays where we do not know the size of the array.

```
program alloc
implicit none
integer, allocatable,dimension(:):: vector
!note syntax - dimension(:)
integer                          :: elements,i
print *,'enter the number of elements in the vector'

read *,elements
allocate(vector(elements))
!allocates the correct amount of memory
print *,' your vector is of size ',elements,'. Now enter each
element'
do i=1,elements
   read *,vector(i)
end do
print *,'This is your vector'

do i=1,elements
   print *,vector(i)
end do

deallocate(vector)
!tidies up the memory

end program alloc
```

The program is called alloc.f95 and can be copied from the web page. Note in particular the bolded lines. The new way of declaring the array **vector** tells the compiler that it is allocatable – ie the size will be determined at **run time.**

We shall look at this further in Section 7.

**Exercise 5.1**

Write a program that asks the user how many numbers they want to enter, call this value **imax**. Allocate **imax** elements to two arrays, **a** and **b**. Read in **imax** numbers to **a** and do the same to **b.** Print out the arrays **a, b** and print out the sum of **a** and **b**. Compare your attempt with sumalloc.f95.

## *5.3 Array magic*

One of the benefits of arrays is that you can easily do operations on every element by using simple arithmetic operators.

```
program ramagic
implicit none
real ,dimension(100) :: a,b,c,d
open(10,file='data.txt')
read(10,*) a
b=a*10
c=b-a
```

```
            d=1
            print *, 'a= ',a
            print *, 'b= ',b
            print *, 'c= ',c
            print *, 'd= ',d
            end program ramagic
```

**Exercise 5.2**

Copy program **ramagic.f95** and file **data.txt** to your own filespace. Run the program and examine the output.

**Exercise 5.3**

Write a program that fills a 10 element array **x** with values between 0 and .9 in steps of .1. Print the values of sin(x) and cos(x) using the properties of arrays to simplify your program. Compare your answer with ramagic2.f95.

## *5.4 Multi dimensional arrays*

The arrays we have looked at so far have been **one dimensional**, that is a single list of numbers that are accessed using a single subscript. In concept, 1 dimensional arrays work in a similar way to vectors. We can also use two dimensional arrays which conceptually are equivalent to matrices.

So, for example,

Integer, dimension(5,5) :: a

sets up a storage space with 25 integer locations.

The next program creates a 2 dimensional array with 2 rows and 3 columns. It fills all locations in column 1 with 1, columns 2 with 2, column 3 with 3 and so on.

```
        program twodra
        implicit none
        integer,dimension(2,3)      :: a
        integer                     ::row,col,count
        count = 0
!creates an array with 3 cols and 2 rows
!sets col 1 to 1, col2 to 2 and so on
        do row=1,2
           count=0
           do col =1,3
                count=count+1
                a(row,col)=count
           end do
        end do
        do row=1,2
              do col =1,3
                   print *,a(row,col)
              end do
        end do
        end program twodra
```

FORTRAN actually allows the use of arrays of up to 7 dimensions, a feature which is rarely needed. To specify a extended precision 3 dimensional array b with subscripts ranging from 1 to 10, 1 to 20 and 1 to 30 we would write:

```
real  (kind=ikind),dimension(10,20,30) :: b
```

## Exercise 5.4

Using a 4*4 array create an identity matrix, that is, a matrix of the form:

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

and output it. **Wouldn't it be nice if we could actually output the matrix elements in rows and columns?** At the end of this section we shall see exactly how to do this.

## *5.5 Formatting your output*

You may now be wondering if there is any way to have better control over what your output looks like. So far we have been using the default output option – that's what the *'s are for in the **write** and **print** statements:

```
write(10,*) x,y,z
print *, 'program finished'
```

## Exercise 5.5

Copy **format.f95,** and run it

```
      program format
      implicit none
!demonstrates use of the format statement
      integer, parameter :: ikind=selected_real_kind(p=15)
      real , dimension(4)               :: x
      integer, dimension(4)             :: nums
   integer                              :: i
      real(kind=ikind),dimension(4)     :: computed
!fill up the arrays with something
      do i = 1,4
          nums(i)           = i * 10
          computed(i)       = cos(0.1*i)
          x(i)              = computed(i)
      end do
      print *,'nums - integer'
      write(*,1) nums
1     format(2i10)
      print *, 'x - real'
      write(*,2) x
2     format(f6.2)
      print *, 'computed - double precision'
      write(*,3) computed
3     format(f20.7)
```

 

```
end program format
```

You can see that the **write** and **format** statements come in pairs.

```
        write(output device,label) variable(s)
label   format(specification)
```

We are using in this example a * as the output device – in other words, the screen.

The **format** statement can actually go anywhere in the program, but by convention we usually place them just after the associated **write** or all together at the end of the program. It's just a matter of taste.

The tricky part here is the **specification**. There are different specifications for integer, real, and character variables.

## 5.5.1 Integer Specification

General form : n**i**m

- Right justified
- m is the number of character spaces reserved for printing (including the sign if there is one)
- If the actual width is less than m, blanks are printed
- n is the number of integers to output per line. If omitted, one number is output per line.

## 5.5.2 Floating point Specification

General form : n**f**m.d

- Right justified
- m is the number of character spaces reserved for printing (including the sign if there is one), and the decimal point.
- If the actual width is less than m, blanks are printed
- n is the number of real numbers to output per line. If omitted, one number is output per line.
- d is the number of spaces reserved for the fractional part of the number – filled with 0's if fewer spaces are needed. If the fractional part is too wide it is rounded.

If the total width for output (m) is too small, FORTRAN will just output *'s.

**Rule** m >= width of the integer part **plus**
d **plus**
1 (space for decimal point) **plus**
1 (space for sign – if negative)

Essentially, make m nice and wide and you won't have any trouble!

## 5.5.3 Exponential Specification

General form nEm.d

- Alternative specification for outputting **real**
- **d** is the number of decimal places
- **m** is the total width of the field including the sign (if any), the character E and its sign, the decimal point and the number of places of decimals. Again make **m** nice and wide to ensure the field is properly printed out.
- n is the number of exponential numbers to output per line. If omitted, one number is output per line.

**Example**

```
          real :: a,b
          a = sqrt(5.0)
          b = -sqrt(a)
          write(*,10) a,b
     10   format(2E14.5)
```

produces:

```
     0.22361E+01  -0.14953E+01
```

### 5.5.4 Character Specification

General form nAm

- n is the number of strings to print
- m is the maximum number of characters to output

**Example:**

```
            program chars
            implicit none
            character ::a*10,b*10
            a='hello'
            b='goodbye'
            write(*,10) a,b
     10     format(2a10)
            end program chars
```

**Exercise 5.6**

Using the format specifications in **format.f95** as a guide, produce a table of

x     $e^x$

where $0 \leq x \leq 1$, for values of x in increments of 0.1. Write your output to a file called myoutput. Ensure that your output lines up neatly in columns. An example program is neatoutput.f95 is available on the website.

## *5.6 Implied Do Loop to write arrays*

So far, the method we have used for input and output of arrays is:

```
        integer :: col,row
        real :: ra(10,10)
     !using do loop
        do row = 1,10
            do col = 1,10
               read *,       ra(row,col)
               write(*,*)    ra(row,col)
            end do
         end do
```

The trouble with this method is that the rows and columns are not preserved on output. An alternative, and neater method is to use **an implied do loop** in the write statement.

```
     real :: ra(10,10)
     integer :: row,col
!use implied do
     do row = 1,10
       do col = 1,10
          read *,        ra(row,col)
       end do
    end do
    do row=1,10
       write(*,10) (ra(row,col),col=1,10)
    end do
10  format(10f5.1)
```

## Exercise 5.7

In Exercise 5.4 you wrote a program to produce and identity matrix. Apply what you know about formatting now to make a neatly formatted matrix onscreen. There is an example identity1.f95 available on the website.