

Project 4 — Arrays and List Sorting

David Arnold

February 26, 2002

1 Introduction

One of the fundamental data constructs in Fortran is the *array*. You can think of an array as a matrix or vector. For example,

$$A = \begin{pmatrix} 3 & -2 & 1 \\ 2 & -2 & 2 \\ 1 & 0 & 0 \end{pmatrix} \quad \text{and} \quad \mathbf{v} = \begin{pmatrix} 1 \\ 2 \\ -3 \end{pmatrix}$$

are arrays; that is, arrangements of numbers. In Fortran, when you declare an array, a contiguous block of memory is set aside to contain the elements of your array.

2 Subscripts

In college algebra and/or calculus, you studied sequences. The notation used for sequences is

$$a_1, a_2, \dots, a_n, \dots$$

In the expression a_n , the symbol n is called a *subscript*, and it is written at the lower right-hand corner of the variable a . The expression a_n represents the n th term of the sequence. For example, if

$$a_n = 2n - 3, \tag{1}$$

then

$$a_{50} = 2(50) - 3 = 97.$$

Note how we substituted 50 for n in formula (1).

In Fortran, subscripts are used in a manner similar to their use with sequences. For example, suppose that the variable A contains an array. Then $A(1)$ would refer to the first element of the array, $A(2)$ would refer to the second element of the array, and so on.

Your subscript can also be an integer variable or expression. For example, $A(I)$ would refer to the I element of the array. Similarly, $A(2 * K - 1)$ refers to the $2k - 1$ st element of the array. By varying the value of I and K in these examples, we can refer to a number of different elements of the array A . We will see that DO loops are one of the most efficient ways to gain access to the elements of an array.

Everyone wants to write code that is portable. It is frustrating to write code that compiles and runs well on your PC, but when you try to compile on another platform, your code is filled with errors. In order to make your code more portable, try to follow these rules when subscripting arrays.

Type of subscript	Example
Integer constant	$A(3)$
Integer variable	$A(I)$
Integer variable + Integer constant	$A(J + 2)$
Integer constant \times Integer variable	$A(3 * K)$
Integer constant \times Integer variable + Integer constant	$A(3 * K + 5)$

Table 1: Permissible subscripts.

3 Declaring Arrays

Declaring arrays is a simple matter. Here is the basic syntax for declaring arrays.

```
type arrayname(LowerLimit:UpperLimit)
```

This syntax warrants explanation. The type of the array are the types that we have been discussing for other variable declarations: integer, real, character, double precision, complex, logical, etc. Next comes the array name. After that, enter the lower and upper bound for the subscripts of the array. For example, the code

```
real x(1:5)
```

declares `x` to be an array that contains real numbers. In addition, the lower bound on subscripts is 1 and the upper bound on subscripts is 5. This means that a contiguous block of memory is reserved for array elements

`x(1)`, `x(2)`, `x(3)`, `x(4)`, and `x(5)`.

The array declaration

```
integer k(-3:3)
```

reserves a continuous block of memory to contain integers

`k(-3)`, `k(-2)`, `k(-1)`, `k(0)`, `k(1)`, `k(2)`, and `k(3)`.

If a lower limit is not declared, then the lower limit is assumed to be 1. For example, the declaration

```
real temp(4)
```

reserves a contiguous block of memory for the real numbers

`temp(1)`, `temp(2)`, `temp(3)`, and `temp(4)`.

You can also have arrays with multiple dimensions. For example, the command (note the comma instead of the colon)

```
integer zeta(3,4)
```

reserves a block of continuous memory to store integers in the 3×4 array `zeta`. In this way, we can refer to elements of the array such as `zeta(1,1)`, `zeta(2,1)`, etc. Indeed, the array `zeta` is actually a matrix having 3 rows and 4 columns. If we want to refer to the entry in the third row and fourth column, then the Fortran syntax `zeta(3,4)` will access this element of the array. Clearly, arrays with multiple dimensions are important, but we will have no need of them in this activity.

4 Populating Arrays With Data

Once you've declared an array variable, you will probably want to store some data in the array. There are several ways that this can be accomplished. You can initialize array entries in your program, you can read array entries from standard input, you can use a data statement, or you can read data from a file, to name a few. Let's begin by writing some code that initializes the array from within the program itself.¹

4.1 Coding Data Into The Array

It's a simple matter to use your code to populate the entries of an array. For example, suppose that you wish to evaluate the function $f(x) = x^2$ for the integers ranging from -3 to 3 .

```
program prog1
  integer fcn(-3:3), i
  do 10, i=-3,3
    fcn(i)=i**2
10  continue
  write(*,*) fcn
end
```

This code warrants some explanation.

- First, note the integer array declaration `integer fcn(-3:3)`. This declares `fcn` as an integer array, with subscripts ranging from -3 to 3 in increments of 1 .
- Next comes a DO loop, where we evaluate the function $f(x) = x^2$ at each integer, starting with -3 and ending with 3 , stepping in increments of 1 .
- Of course, we would like to see if the array actually contains the expected results, so the write statement `write(*,*) fcn` prints the contents of the array to the screen as an unformatted list. This is the simplest way to display the contents of an array. We will explore other methods in upcoming examples.

The output of this routine looks like the following:

```
prog1
 9 4 1 0 1 4 9
```

4.2 Populating Arrays From The Standard Input

You can also initialize each array entry from the standard input, much as you've read data from the standard input (usually the keyboard) in your previous programs. Simply write the appropriate DO loop.

```
program prog2
  real temp(5)
  integer i
  do 10, i=1,5
```

¹The code that follows is complete. I strongly urge you to give these programs a try as you read through this activity. Type them in, compile them, then execute them. There is a lot to be learned when you do this.

```

        read(*,*) temp(i)
10  continue
    write(*,*)
    write(*,'(1x,2A10)') 'Reading', 'Temp'
    do 20, i=1,5
        write(*,'(1x,I10,F10.1)') i, temp(i)
20  continue
    end

```

Again, this code deserves some explanatory comments.

- We first declare a real array with `real temp(5)`. This gives us a real array with an index starting at 1 and ending with 5, in increments of 1. The intent here is that this array will hold 5 temperature measurements, which will be read from the standard input (in this case the keyboard).
- A DO loop, couple with a read statement `read(*,*) temp(i)`, populates the array with temperature readings entered from the keyboard. This interaction will be evident in the program run shown below.
- Finally, format statements provide a nicer looking output generated, again, with the aid of a DO loop.

A sample program run follows.

```

prog2
10.2
22.3
33.9
44.2
51.8

```

Reading	Temp
1	10.2
2	22.3
3	33.9
4	44.2
5	51.8

4.3 Using Data Statements

You've probably experienced the pain of entering data at the keyboard when constructing and debugging your programs. It can be a real pain to enter data in this manner, particularly at the development stage of your program. Well, we have good news for you. The `DATA` statement will cure your blues in this regard.

In its simplest form, you can initialize a single variable.

```

integer i
data i/20/

```

This code snippet first declares the variable `i` to be an integer. Then, the data statement, `data i/20/` initializes `i` to equal 20. This data statement is almost identical to the statement `i=20`, so you might question its value, but be patient. You'll soon see the power of data statements.

For example, you can initialize a number of different variables as follows.

```

integer i
real a, b, c
data i, a, b, c/12, 1.0, 3.4, 5.6/

```

Ah! Things are looking up. Obviously, the ability to initialize a host of variables simultaneously can be useful. Things even look better when you see that you can use data statements to initialize arrays.

```

program prog3
real temp(5)
integer i
data temp/10.2,22.3,33.9,44.2,51.8/
write(*,*)
write(*,'(1x,2A10)') 'Reading', 'Temp'
do 20, i=1,5
    write(*,'(1x,I10,F10.1)') i, temp(i)
20 continue
end

```

Clearly, populating an array with a data statement is far superior to entering the data over and over again from the keyboard during program development, as can be clearly seen from the output of this program.

prog3

Reading	Temp
1	10.2
2	22.3
3	33.9
4	44.2
5	51.8

It is true, however, that some circumstances might require that you put the read loop back into your program once the development phase has finished. In this way, the user of your program can now enter the data from the keyboard if that is requested in the program design.

4.4 Entering Data From A File

Sometimes, the data set you will use already exists. Perhaps you have a set of financial data that you've downloaded from the internet and have saved in a data file. Then, the most efficient way to populate your array with this data is to read the data into your array directly from the file. After all, who wants to sit at a terminal and enter thousands of lines of financial data? :-)

So, let us imagine that we have financial data stored in a file named `data.txt`. For purposes of exposition, we have created a file called `data.txt` with the following 10 entries.

```

156.64
178.87
233.42
196.45
234.56
789.99

```

```
123.32
456.32
222.22
256.78
```

4.4.1 The CRLF Issue

Now, if we were working strictly in UNIX or Linux, we wouldn't have any worries. But we are working with a windows version of NTEmacs, and Cygwin is a windows port of the UNIX operating system, so we are going to encounter the CRLF issue that is encountered when transferring files from one operating system to another.

To make the story short and to the point, different operating systems use different characters to indicate the end of a line. These characters are put into your file each time you hit the Enter key to terminate the current line and begin another line. Some operating systems just use a carriage return (CR) when you hit the Enter key, others use just a line feed (LF) when you hit the Enter key, and others use both (CRLF) when you hit the Enter key. This makes porting your files from one operating system to another difficult, to say the least.

After you enter the data above in NTEmacs, save the file as `data.txt`, then open Cygwin and type the command:

```
cat -A data.txt
156.64^M$
178.87^M$
233.42^M$
196.45^M$
234.56^M$
789.99^M$
123.32^M$
456.32^M$
222.22^M$
256.78
```

The `cat` command is a UNIX command that lists the contents of the file. The `-A` switch tells it to display hidden characters in the file. The character `^M$` is the end-of-line character in the DOS operating system (Windows). Note that there is no character after the last entry (256.78), which is going to cause a problem when reading this data file with Fortran. Return to the file `data.txt` and press Enter after this last entry. Now, the `cat` command shows an end-of-line character after the last entry.

```
cat -A data.txt
156.64^M$
178.87^M$
233.42^M$
196.45^M$
234.56^M$
789.99^M$
123.32^M$
456.32^M$
222.22^M$
256.78^M$
```

In order to use the file `data.txt` with the Cygwin compiler `g77`, which is a UNIX tool, we are going to have to change all of the end-of-line characters in the file `data.txt` to UNIX end-of-line characters.

NTEmacs provides an easy tool that allows us to change the encoding of our data file. Reopen the file `data.txt`. Then press `C-x` and hit the Enter key. Next, hit the `f` key on your keyboard. You will be prompted with “Coding system for visited file (default, nil):”, at which point you enter “unix” and press the Enter key. Save your file. Now, return to Cygwin and retry the `cat` command.

```
cat -A data.txt
156.64$
178.87$
233.42$
196.45$
234.56$
789.99$
123.32$
456.32$
222.22$
256.78$
```

Note how all of the end-of-line characters are now changed to the UNIX end-of-line characters. This file is now suitable for calling with `g77`.

4.4.2 Reading File Data Into An Array

Now that we’ve crafted the file `data.txt`, let’s write a program that will read data from this file into an array.

```
program prog4
real paymen(10)
integer i
open(unit=1,file='data.txt',status='old')
do 10, i=1,10
    read(1,*) paymen(i)
10 continue
write(*,*)
write(*,'(1x,2A10)') 'Number', 'Payment'
do 20, i=1,10
    write(*,'(1x,I10,F10.1)') i, paymen(i)
20 continue
close(1)
end
```

Again, some comments are in order.

- First, we declare an array variable to hold payment data with `paymen(10)`. This creates a real array to hold 10 entries, with the index running from 1 to 10 in increments of 1.
- Next, the command

```
open(unit=1,file='data.txt',status='old')
```

opens the file `data.txt` and associates it with the file number 1. After that, any reference to the number 1 is a reference to the file `data.txt`. The file is opened with `status='old'` because the file is an existing ('old') file from which we wish to read data.

- Next, a DO loop is written to read 10 records from the file. Each line of a file contains a "record." If the data in the file was arranged all on one line, as in

156.6, 178.9, ..., 256.8

then a different strategy would be required for reading the file. This is because of the fact that every time we execute `read(1,*) paymen(i)`, a new record (line) of the file is read, and the first value found on the new line is stored in the array entry `paymen(i)`.

We will have more to say about reading data from files as the class proceeds. However, for now, consider only this case, as this is how I've arranged the data in the file used in this activity.

5 Program 4

In this activity, you are to read in the examination scores from a file named `scores.txt`. There are 100 examination scores in the file `scores.txt`, arranged one examination score per record (per line). The first step is to download the file `scores.txt` from the following url.

<http://online.redwoods.cc.ca.us/instruct/darnold/Fortran/Activities/SortingLists/scores.txt>

Next, write subroutines to perform each of the following tasks.

1. Read the data in `scores.txt` into an array.
2. Find the sum of the elements of your array.
3. Find the average of the elements in your array.
4. Sort the array elements in descending order (highest value first, lowest value last).
5. Print the results.

If you are thinking modularly, then your main program will look like this:

```
program program4
  real scores(100), sum, avg
  integer n
  parameter(n=100)
  call read_data(n,scores)
  call sum_array(n,scores,sum)
  call average_array(n,scores,avg)
  call sort_array(n,scores)
  call print_results(n,scores,sum,avg)
end
```

You need to write five subroutines, `read_data`, `sum_array`, `average_array`, `sort_array`, and the `print_results` subroutine.

5.1 Passing Arrays to Subroutines

The careful reader will note that we are passing entire arrays to our subroutines. For example, in the call

```
call sum_array(n,scores,sum)
```

the variable `scores` contains a real array of examination score data. An immediate question arises: How do we craft our subroutine so that it will handle arrays of various sizes?

You first reaction might be “who cares?,” but a careful consideration of the role of the subroutine `sum` will quickly change your mind. The idea is simple. You want your subroutine `sum` to find the sum of the elements of any array you might send to the routine. After all, subroutines are supposed to be pieces of code that can be reused in any number of routines. That is, if you have a program that needs to find the sum of the elements of an array, you don’t want to rewrite the subroutine just to fit the parameters of the new program. You want a subroutine that will sum the elements of an array in all situations.

So, that said, how do we write our subroutine to allow for the input of arrays of various sizes? The answer is also simple, as Fortran allows us to declare arrays somewhat “dynamically” in subroutines. This is not allowed in the main program, but it is allowed in subroutines. The reason for this is clear. In the main program, you declare space for the array when you declare it. When you pass the array to the subroutine, you are just passing an address for the array, you are not creating a new array. So, all the subroutine needs to know is the size of the array. It doesn’t need to reserve space for the array, as that has already been done in the main program.

This is precisely why we not only pass the array to the subroutine, we also pass the size of the array in the call

```
call sum_array(n,scores,sum)
```

Therefore, we begin our subroutine as follows.

```
subroutine sum_array(m,x,s)
  integer m
  real s
  real x(m)
```

That’s all there is to it! You now craft the remainder of the subroutine in the usual manner.

5.2 Wrapping It Up

Arrange your write statements so that output is sent to the screen. On this project, I don’t care to collect printed output of summary results, so it is not necessary to print your output to a file for eventual printing. However, you might think about a nice way of arranging your output so that all the results will fit on one screen. That would be helpful when I run your program during the grading cycle.

Save and compile your program, then, once it’s running properly, obtain a hardcopy printout for submission. In addition, place all files associated with this project in your Fortran folder in a folder called Project4 on your H drive on the school’s server. I will test each of your programs for accuracy from my office, but I won’t be able to locate your files unless you follow this convention.

6 The Grading Rubric

The following rules will apply for this program, after which we will discuss and adjust the rubric during class.

1. (30 points) Will be awarded for adequate comments. Comments should include:
 - (a) A description of the program's purpose.
 - (b) Name, date, version or revision number.
 - (c) A complete dictionary of all variables and parameters used in the program.
 - (d) Interprogram comments should precede any code snippets explained by the comments. These should be adequately sprinkled throughout your code.
2. (50 points) Will be awarded if the program works and does what it was asked to do.
3. (10 points) Will be awarded for good program style. This includes good indentation practices, etc.
4. (10 points) Will be awarded for creativity and extra effort. Did you just do the bare minimum? Or did you stretch and reach a little higher? Did you put something cute or clever into your program that nobody else seemed to think of?

7 Penalties

Each program that is assigned during the term will have a due date. On that date, the program must be on the instructor's desk before the start of class. Penalties will be assessed as follows.

1. (10 points) There will be a 10 point deduction for any program that is handed in after the class has begun.
2. (20 points) There will be a 20 point deduction per class period. That is, if you hand the program in one class period late, there is an automatic 20 point deduction. Two class periods warrants a 40 point deduction, etc. To be clear, if the program is in the instructor's hands before the beginning of the next class, that is a 20 point deduction. If the program is in the instructor's hands before the start of the second class period past the due date, that is a 40 point deduction, etc.

8 Managing Files and Folders

Each of you has been given personal space on the sci-math server to store your work. Typically, this space is mapped to the drive letter H. If you open the Windows Explorer (the file manager, not the internet browser), you can see that the drive letter has been mapped to your login name.

In this folder, create a new folder call FortranPrograms. Note that you must **never** use spaces in filenames. In the Windows operating system, filenames are not case-sensitive, which is exactly opposite what happens in Unix and Linux, where filenames are case-sensitive.

In your FortranPrograms folder, create another folder called Program4. It is in this folder that you are to place the source code and executables for this current project. When you receive your next project, create a new folder called Program4 to hold that project, etc.

If you work at home, I still want you to place copies of your work in the space reserved for you on our system. Simply copy your home files onto a floppy disk and bring them with you to school. Use the Windows Explorer to copy the files on your disk into the proper folder (H:/FortranPrograms/Program4).

If everyone follows these simple rules, I can easily access your work from my office machine for purposes of assigning a grade.

9 Caveat

On this project, if you stop by my office with hardcopy of your program before the due date of this assignment, I will give a quick glance and critique of your source code. Somewhat like receiving a grade on a draft before submitting your final draft for assessment.