FORTRAN LESSON 5

Lesson Topics	
Arrays	Factorials
Dimension Statement	Arrays in Function Subprograms
Function Subprograms	Return in Function Subprograms
View Demos	**Download Demos**
# 1 # 2 # 3	# 1 # 2 # 3
Main Fortran Page	

Arrays

There are only a few minor differences in the way Fortran and Basic treat arrays. Array declarations in Fortran go at the beginning of the program, before any executable statement. Arrays can be declared with either a *dimension* statement or a type declaration. The latter way is preferred, because it is best anyway to declare the type of the array. Here are examples of arrays introduced by type declarations:

real a(10), b(5)	one-dimensional arrays a and b of real variables, indexed from 1 to 10 and from 1 to 5, respectively
integer n(3:8), m	one-dimensional array n of integers, indexed from 3 to 8, and an integer variable m
double precision c(4,5)	two-dimensional array c of double precision real numbers, the first index running from 1 to 4, and the second from 1 to 5
character student(30)*20	one-dimensional array *student* of strings, indexed from 1 to 30, each string up to 20 symbols long
real num(0:5,1:10,-3:3)	three-dimensional array *num* of single precision real numbers, the first index running from 0 to 5, the second from 1 to 10, and the third from -3 to 3

In Fortran the default lower limit of the range of a subscript is 1, rather than 0 as in Basic. A colon separates the lower and upper limits whenever both are specified.

Because arrays are declared at the beginning of the program, they must be given a fixed size - i.e., the limits must be constants rather than variables. (In this respect Fortran is less flexible than Basic, in that Basic allows the dimension of an array to be a variable whose value can be input by the user, thereby ensuring that exactly the right amount of storage space is reserved.) You don't have to use the full size of the array specified in the declaration statements; that is, you may reserve space for more entries in the array than you will need.

If you use a dimension statement to declare an array, you should precede it with a type declaration. Here is one way to introduce a real array *weights*, indexed from 1 to 7:

```
real weights
dimension weights(7)
```

But the same can be accomplished more briefly with the single statement

```
real weights(7)   .
```

Although the upper and lower limits of an array cannot be variables, they can be constants declared in parameter statements. The sequence of statements

```
integer max
parameter (max = 100)
character names(max)*30
real scores(max)
```

instructs Fortran to set aside storage space for a list of at most 100 names, each a string of length no longer than 30 symbols, as well as a list of at most 100 scores, each a real number.

As in Basic, in Fortran you may input and print arrays with do loops. But you can sometimes more efficiently do the same with single statements. For instance, the above array *weights* can be input with only the statement

```
read *, weights   .
```

This *read* statement pauses the program to allow the user to enter all seven entries of the array. The user can either enter the seven weights one-by-one separated by returns, or alternatively, can enter all seven weights separated only by commas, and then a single return. If you want to input say only the first five weights, you can do so with the statement

```
read *, (weights(i), i=1,5)   .
```

Analogously, the single print statement

```
print *, weights
```

prints the seven entries of *weights* to the screen, while the statement

```
print *, (weights(i), i=p,q)
```

prints only the weights indexed from p to q.

There are various formatting tricks useful in printing two-dimensional arrays. Here is one example demonstrating how to print a matrix A having 5 rows and 6 columns of real numbers, with each row of the matrix printed on its own line :

```
do i = 1, 5
        write (*,10) (A(i,j), j = 1, 6)
end do
```

```
10          format (6f7.3)
```

More precise formatting can be accomplished with double loops and tab indicators.

Function Subprograms

Function subprograms in Fortran define functions too complicated to describe in one line. Here is a function subprogram defining the factorial function, fact(n) = n! :

```
function fact(n)
integer fact, n, p
p = 1
do i = 1, n
        p = p * i
end do
fact = p
end
```

The first line of the function subprogram specifies the name of the function, and lists in parentheses the variables upon which the function depends. The subprogram has its own type statements, declaring the type of the function itself, as well as the types of the variables involved in computing the function. Somewhere in the subprogram there must be a line giving the value of the function. (Above it is the line "fact = p".) The subprogram concludes with an *end* statement. In Fortran, function subprograms do not have to be declared as they do in Basic. The entire function subprogram appears in the source file after the final *end* statement of the main program.

The above factorial subprogram, with variables of integer type, works only for nonnegative integers no larger than 12, as 13! = 6,227,020,800 exceeds the Fortran upper limit of 2,147,483,647 for integers. To handle larger integers, the types can be changed to real or double precision. In GNU Fortran, single precision real type handles factorials of integers as large as 34, and double precision as large as 170.

The main program (or in fact any subprogram) utilizing a function subprogram should likewise specify the type of the function. Here is a simple main program using the above factorial function "fact":

```
program demofactorial
integer fact, n
print *, "What is n?"
read *, n
print *, "The value of", n, " factorial is", fact(n)
end
```

Because n is declared an integer in the function subprogram defining fact(n), it must also be an integer in the main program when fact(n) is evaluated; if it is of a different type the compiler displays a type mismatch error message.

A function subprogram may depend on several variables, and it may use an already defined statement function or a function defined by another function subprogram. Following is a function subprogram utilizing the above factorial function subprogram; it computes the Poisson probability function, defined as

$$P(n,t) = t^n e^{-t} / n! \quad ,$$

where n is a nonnegative integer and t any positive number:

```
function poisson(n,t)
real poisson, t
integer n, fact
poisson = (t ** n) * exp(-t) / fact(n)
end
```

Note that, as this subprogram references the function "fact", it must declare its type. Both this subprogram and the factorial subprogram will appear in the source file following the *end* statement for the main program. (The order in which the subprograms are typed makes no difference - just as long as they both follow the main program.)

Again, in referencing function subprograms one must respect types; for example, if the main program is to compute poisson(m,s) for some variables m and s, then, in order to conform to the type declarations in the function poisson, m must first be declared an integer and s of real type. Oversights will lead to compiler type-mismatch messages.

Arrays in Function Subprograms

An array can be listed as a variable of a function defined by a function subprogram - but you just write the array name, with no parentheses after the name as in Basic. The type and dimension of the array must be specified in the function subprogram.

Following is a program called "mean" that computes the mean, or average, of a list containing up to 100 numbers. The main program prompts for the list of numbers, and then references a function subprogram named "avg" that computes the average.

```
program mean
real numbers(100), avg
integer m
print *, "How many numbers are on your list?"
print *, "(no more than 100, please)"
read *, m
do i =1, m
        print *, "Enter your next number:"
        read *, numbers(i)
end do
print *, "The average is", avg(m,numbers)
end

function avg(n,list)
real avg, list(100), sum
integer n
sum = 0
do i = 1, n
        sum = sum + list(i)
end do
avg = sum/n
```

```
end
```

Note that both the main program and the subprogram declare the type of the function "avg". The main program calls the function subprogram with the arguments "m" and "numbers", and these are substituted into the function subprogram for the variables "n" and "list". The main program specifies the dimension of the array "numbers", while the subprogram specifies the dimension of the array "list". The subprogram does its calculations and returns the value of "avg" to the main program. For this procedure to work, the types of the variables "m" and "n" must agree, as well as the types of "numbers" and "list".

Return (in Function Subprograms)

A *return* statement in a function subprogram acts like a *stop* statement in the main program; the subprogram is terminated and Fortran returns to where it left off in the main program. Here is a function subprogram defined on integers n; the value of the function "demo" is 0 if n ≤ 0, and if n > 0 it is the sum of the squares of the integers from 1 to n:

```
function demo(n)
integer demo, n
demo = 0
if (n .le. 0) return
do i =1, n
        demo = demo + i * i
end do
end
```