5 Fortran Arrays

Fortran arrays are any object with the dimension attribute. In Fortran 90, and in HPF, arrays may be very different from arrays in older versions of Fortran. Arrays can have values assigned as a whole without specifying operations on individual array elements, and array sections can be accessed. Also, allocatable arrays that are created dynamically are available in Fortran 90. Arrays in HPF play a central role in data distribution and data alignment (refer to *The pghpf User's Guide* and *The High Performance Fortran Handbook* for details on working with arrays in HPF). This chapter describes some of the features of Fortran 90/HPF arrays.

The following example illustrates valid array operations.

```
REAL(10,10) A,B,C
A=12            !Assign 12 to all elements of A
B=3             !Assign 3 to all elements of B
C=A+B           !Add each element of A to each of B
```

5.1 Array Types

Fortran supports four types of arrays *explicit-shape* arrays, *assumed-shape* arrays, *deferred-shape* arrays and *assumed-size* arrays. Both explicit-shape arrays and deferred shape arrays are valid in a main program. Assumed shape arrays and assumed size arrays are only valid for arrays used as dummy arguments. Deferred shape arrays, where the storage for the array is allocated during execution, must be declared with either the ALLOCATABLE or POINTER attributes.

Every array has properties of type *rank*, *shape* and *size*. The *extent* of an array's dimension is the number of elements in the dimension. The array rank is the number of dimensions in the array, up to a maximum of seven. The shape is the vector representing the extents for all dimensions. The size is the product of the extents. For some types of arrays, all of these properties are determined when the array is declared. For other types of arrays, some of these properties are determined when the array is allocated or when a procedure using the array is entered. For arrays that are dummy arguments, there are several special cases.

Allocatable arrays are arrays that are declared but for which no storage is allocated until an allocate statement is executed when the program is running. Allocatable arrays provide Fortran 90 and HPF programs with dynamic storage. Allocatable arrays are declared with a rank specified with the ":" character rather than with explicit extents, and they are given the ALLOCATABLE attribute.

5.1.1 Explicit Shape Arrays

Explicit shape arrays are those arrays familiar to Fortran 77 programmers. Each dimension is declared with an explicit value. There are two special cases of explicit arrays, in a procedure, an explicit array whose bounds are passed in from the calling program are called *automatic-arrays*. As a second type, in a procedure, where an array is a dummy array and the bounds are passed from the calling program, is called an *adjustable-array*.

5.1.2 Assumed Shape Arrays

An assumed shape array is a dummy array whose bounds are determined from the actual array. Intrinsics called from the called program can determine sizes of the extents in the called program's dummy array.

5.1.3 Deferred Shape Arrays

A deferred shape array is an array that is declared, but not with an explicit shape. Upon declaration, the array's type, its kind, and its rank (number of dimensions) are determined. Deferred shape arrays are of two varieties, *allocatable arrays* and *array pointers*.

5.1.4 Assumed Size Arrays

An assumed size array is a dummy array whose size is determined from the corresponding array in the calling program. The arrays rank and extents may not be declared the same as the original array, but its total size (number of elements) is the same as the actual array. This form of array should not need to be used in new Fortran programs.

5.2 Array Specification

Arrays may be specified in either of two types of data type specification statements, attribute-oriented specifications or entity-oriented specifications. Arrays may also optionally have data assigned to them when they are declared. This section covers the basic form of entity-based declarations for the various types of arrays. Note that all the details of array passing for procedures are not covered in this Chapter (refer to *The Fortran 90 Handbook* for complete details on the use of arrays as dummy arguments).

5.2.1 Explicit Shape Arrays

Explicit shape arrays are defined with a specified rank, each dimension must have an upper bound specified, and a lower bound may be specified. Each bound is explicitly defined with a specification of the form:

```
[lower-bound:] upper-bound
```

An array has a maximum of seven dimensions. The following are valid explicit array declarations:

```
INTEGER NUM1(1,2,3)              !Three dimensions
INTEGER NUM2(-12:6,100:1000)     !Two dimensions with
INTEGER NUM3(0,12,12,12)!Array of size 0
INTEGER NUM3(M:N,P:Q,L,99)       !Array with 4 dimensions
```

5.2.2 Assumed Shape Arrays

An assumed shape array is always a dummy argument. Assumed shape array has a specification of the form:

```
[lower-bound] :
```

The number of colons (:) determines the array's rank. An assumed shape array cannot be an ALLOCATABLE or POINTER array.

5.2.3 Deferred Shape Arrays

An deferred shape array is an array pointer or an allocatable array. Assumed shape array has a specification determines the array's rank and has the following form for each dimension:

```
:
```

For example:

```
INTEGER, POINTER ::NUM1(:,:,:,:)
INTEGER, ALLOCATABLE::NUM2(:)
```

5.2.4 Assumed Size Arrays

An assumed size array is a dummy argument with an assumed size. The array's rank and bounds are specified with a declaration that has the following form:

```
[explicit-shape-spec-list ,][lower-bound :]*
```

For example:

```
SUBROUTINE YSUM1(M,B,C)
     INTEGER M
     REAL, DIMENSION(M,4,5,*) :: B,C
```

5.3 Array Subscripts and Access

There are a variety of ways to access an array in whole or in part. Arrays can be accessed, used, and assigned to as whole arrays, as elements, or as sections. Array elements are the basic access method, for example:

```
INTEGER, DIMENSION(3,11) ::  NUMB
NUMB(3,1)=5
```

This assigns the value 5 to element 3,1 of `NUMB`

The array `NUMB` may also be accessed as an entire array:

```
NUMB=5
```

This assigns the value 5 to all elements of `NUMB`.

5.3.1 Array Sections and Subscript Triplets

Another possibility for accessing array elements is the *array section*. An array section is an array accessed by a subscript that represents a subset of the entire array's elements and is not an array element. An array section resulting from applying a subscript list may have a different rank than the original array. An array section's subscript list consists of subscripts, *subscript triplets*, and/or *vector subscripts*. For example using a subscript triplet and a subscript:

```
NUMB(:,3)=6
```

assigns the value 6 to all elements of `NUMB` with the second dimension of value 3 (`NUMB(1,3)`, `NUMB (2,3)`, `NUMB(3,3)`). This array section uses the array subscript triplet and a subscript to access three elements of the original array. This array section could also be assigned to a rank one array with three elements, for example:

```
INTEGER(3,11) NUMB
INTEGER(3) NUMC
NUMB(:,3)=6
NUMC=NUMB(:,3)
```

Note that `NUMC` is rank 1 and `NUMB` is rank 2. This array section assignment illustrates how `NUMC`, the array section of `NUMB` has a shape that due to the use of the subscript 3, is of a different rank than the original array.

The general form for an array's dimension with a vector *subscript triplet* is:

```
[subscript] : [subscript] [:stride]
```

The first subscript is the lower bound for the array section, the second is the upper bound and the third is the stride. The stride is by default one. If all values except the `:` are omitted, then all the values for the specified dimensions are included in the array section. For example, using `NUMB` above:

```
NUMB(1:3:2,3)=7
```

assigns the value 7 to the elements `NUMB(1,3)` and `NUMB(3,3)`.

5.3.2 Array Sections and Vector Subscripts

Vector-valued subscripts specify an array section by supplying a set of values defined in a one dimensional array (vector) for a dimension or several dimensions of an array section. For example:

```
INTEGER J(2), I(2)
INTEGER NUMB(3,6)
I=(/1,2/)
J=(/2,3/)
NUMB(J,I)=7
```

This array section uses the vectors I and J to assign the value 7 to the elements `NUMB(2,1)`, `NUMB(2,2)`, `NUMB(3,1)`, `NUMB(3,2)`.

5.4 Array Constructors

An array constructor can be used to assign values to an array. Array constructors form one-dimensional vectors to supply values to a one-dimensional array, or one dimensional vectors and the RESHAPE function to supply values to arrays with more than one dimension.

Array constructors can use a form of implied DO similar to that in a DATA statement. For example:

```
INTEGER DIMENSION(4):: K = (/1,2,7,11/)
INTEGER DIMENSION(20):: J = (/(I,I=1,40,2)/)
```

5.5 PGI Array Extensions (CM Fortran) @

5.5.1 The ARRAY Attribute (CM Fortran) @

PGI provides several extensions for handling arrays. The compiler handles the CM Fortran attribute ARRAY. The ARRAY attribute is similar to the DIMENSION attribute. Refer to Chapter 3, "Fortran Statements" for more details on the ARRAY statement.

5.5.2 Array Constructors Extensions (CM Fortran) @

The PGI compiler supports an extended form of the array constructor specification. In addition to the `(/ ../)` specification for array constructors, PGI supports the following notation where `[` and `]` begin and end, respectively, an array constructor.

In addition, an array constructor item may be a 'subscript triplet' in the form of an array section where the values are assigned to the array:

```
lower-bound : upper-bound [ : <stride> ]
```

For the values `i : j : k` the array would be assigned values `i, i+k, i+2k, ..., j.` If `k` is not present, stride is assumed to be 1.

For example:

```
INTEGER, DIMENSION(20):: K = [1:40:2]
```