Chapter 4

Array Declarations and Initial Values

CM Fortran supports all the Fortran 77 syntax for declaring and initializing arrays. Any array that is declared and/or initialized in the Fortran 77 manner can be used in an array operation. It is this use — regardless of the declaration syntax — that causes the array to be allocated in CM memory and processed in parallel.

This chapter introduces a number of additional CM Fortran features, many drawn from Fortran 90, for declaring and initializing arrays and, in some cases, scalars as well. These features include:

- *Attributed type declarations,* which specify not only an object's type but also such attributes as **`DIMENSION`**, **`DATA`**, or **`PARAMETER`**. These attributes substitute for a separate **`DIMENSION`**, **`DATA`**, or **`PARAMETER`** statement.
- Fortran 90 *kind* syntax for specifying the precision of numeric types.
- *Array constructors,* which specify the values of a 1-dimensional array. An array constructor can be used, for example, to initialize a vector or to pass a vector of values as an argument to an intrinsic function or external procedure.
- Dynamic initialization by means of array assignments and a CM Fortran utility routine for generating random numbers.

This chapter focuses on declaring and setting the initial values of static local arrays, including arrays that are local to the main program and passed as actual arguments. Two related subjects are deferred until later chapters:

- Common arrays are discussed along with subroutines in Chapter 6.
- Dynamic array allocation is discussed in Chapter 7.

Copyright © 1989–1994 Thinking Machines Corporation

4.1 Declaring Arrays

A Fortran 90 attributed type declaration is a single statement that can associate several attributes besides type with the object being declared. One of these attributes is **DIMENSION**.

NOTE: CM Fortran also accepts the attribute **ARRAY**; **ARRAY** is an earlier name for the **DIMENSION** attribute.

4.1.1 Using the DIMENSION Attribute

An attributed type declaration with the attribute **DIMENSION** has the form:

type , **DIMENSION (*dims*)** :: *name* [, *name* , ...]

For example, the declaration

```
INTEGER, DIMENSION(10) :: ARRAY_NAME
```

is equivalent to the two Fortran 77 statements

```
INTEGER ARRAY_NAME
DIMENSION ARRAY_NAME(10)
```

Attributed type declarations are a convenient way to declare several conformable arrays:

```
REAL, DIMENSION(512) :: A,B,C,D   ! conformable vectors

COMPLEX, DIMENSION(100,100) :: E,F,G,H  ! conformable
                                        ! matrices
```

4.1.2 Array Properties

Declarations of the **DIMENSION** attribute describe two properties of the arrays:

- *rank* (the number of dimensions)
- *shape* (the extents of the dimensions in rank order)

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

Scalars have rank 0, and arrays can have rank 1 through 31. Vectors **A**, **B**, **C**, and **D** shown above have shape [512]; matrices **E**, **F**, **G**, and **H** all have shape [100,100]. The *size* of an array is the product of the extents of its dimensions. Thus, the vectors are all of size 512, and the matrices are of size 10,000.

As in Fortran 77, the extent of a dimension can be specified with a lower bound as well as an upper bound. For example:

```
INTEGER, DIMENSION(-40:100) :: CTEMP
```

This statement declares an array element for each Celsius temperature between the point where the Celsius and Fahrenheit scales converge and the boiling point.

4.1.3 Object Types and Kinds

A third attribute of arrays (and also of scalars) is their type and *kind type* (precision). These can be specified by the appropriate keywords:

CHARACTER	**REAL**
LOGICAL	**DOUBLE PRECISION** or **REAL*8**
INTEGER	**COMPLEX**
INTEGER*8	**DOUBLE COMPLEX**

Recall from Chapter 2 that the **INTEGER*8** type is supported only under the vector-units execution models (**-vu** and **-vu -node**).

Alternatively, you can specify just the base type (**INTEGER**, **REAL**, **COMPLEX**) and use the keyword **KIND=** to indicate the desired precision. The value of **KIND=**, called the kind type parameter, must be a scalar integer initialization expression, meaning a compile-time constant. In CM Fortran, it can be any constant expression that evaluates to 32 or 64. For example:

```
INTEGER(KIND=64) A
INTEGER(KIND=_DOUBLE_INT_) B
REAL(KIND=_DOUBLE_) C

INTEGER LL
PARAMETER (LL=64)
INTEGER(KIND=LL) D
REAL(KIND=LL) E
```


Copyright © 1989–1994 Thinking Machines Corporation

CM Fortran provides predefined symbolic names for the kind types. These are shown below with their values. Notice that for complex types, the length is that of the base type (real or double).

Data Type	Length	Symbolic Name
LOGICAL	32	(none)
INTEGER*4	32	_SINGLE_INT_
INTEGER*8	64	_DOUBLE_INT_
REAL*4	32	_SINGLE_
REAL*8	64	_DOUBLE_
COMPLEX*8	32	_SINGLE_COMPLEX_
COMPLEX*16	64	_DOUBLE_COMPLEX_

Integer and floating-point values default to the 32-bit length unless otherwise declared. The **cmf** compiler provides an option, **-double_precision**, that causes **REAL** values only to default to the 64-bit length. This option does not affect **INTEGER** or **COMPLEX** types.

4.2 Working with Constants

Like Fortran 77, CM Fortran supports declaring named constants, or *parameters*. Like Fortran 90, CM Fortran also supports specifying the kind types of literal constants.

4.2.1 Defining Named Constants

A type declaration with the attribute **PARAMETER** defines a named constant. Like the statement **PARAMETER**, this attribute can be used only for scalar objects, not for arrays.

A declaration with the attribute **PARAMETER** has the form

type **, PARAMETER** **::** *name* **=** *constant-expression*

For example:

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

```
INTEGER, PARAMETER :: ONE = 1
REAL, PARAMETER    :: PI = 22.0 / 7.0
LOGICAL, PARAMETER :: T = .TRUE., F = .FALSE.
```

4.2.2 Specifying Typed Constants

To specify the type, including kind, of a literal constant, append an underscore and then an integer constant expression that evaluates to a valid **KIND** number (32 or 64). For example, a constant of type **REAL(KIND=_DOUBLE_)** could be written either as **1.0D0** or as **1.0__DOUBLE_**.

Note the double underscore when the appended underscore of the typed constant is combined with the prepended underscore of the CM Fortran symbolic name for the kind type parameter:

```
INTEGER LL
PARAMETER (LL=64)
INTEGER(KIND=LL) D
D = 1234_64 + 1234_LL + 1234__DOUBLE_INT_
```

Given the rules of implicit coercion in expressions (described in the *CM Fortran Language Reference Manual),* a typed constant provides a way to coerce variables to the desired length. For example, suppose you want to coerce a 32-bit integer N to length 64:

```
N = N + 0_64
```

An alternative for integer values is to use the intrinsic functions **INT** and **NINT**, which take an optional **KIND** argument:

```
INTEGER*8 BIGINT
INTEGER*4 SMALLINT
...
BIGINT = INT(SMALLINT,KIND=64)
...
BIGINT = INT(SMALLINT,KIND=KIND(BIGINT)) !KIND intrinsic
```

Another example uses a conversion function to move array **SMALLINT** into the high 32 bits of **BIGINT**. The following construction does *not* express this intention; the shift is performed in 32 bits before the coercion to 64 bits.

```
BIGINT = ISHFT( SMALLINT, +32 )  ! Shift before coercion
```

The desired behavior (coerce and then shift by 32 positions) is expressed by:

Copyright © 1989–1994 Thinking Machines Corporation

```
BIGINT = ISHFT( INT( SMALLINT,KIND=64 ),+32 )
```

The comparable expression using a typed constant is:

```
BIGINT = ISHFT( SMALLINT+0_64, +32 )
```

4.3 Setting Array Values

Arrays can be initialized with a **DATA** statement, as in Fortran 77, or with the **DATA** attribute in a type declaration. The values of a **DATA** attribute for an array are specified with a CM Fortran feature called an *array constructor*.

As with declaration syntax, the form chosen for static initialization does not affect the home of an array or the operations that may be performed on it.

Initial array values can also be set dynamically by means of assignment. CM array objects can be assigned array constructors, or they can be seeded with random values by means of a CM Fortran utility routine. These operations cannot be performed on front-end arrays.

4.3.1 Assignment

As in Fortran 77, arrays can be initialized dynamically by assigning a value to each element. However, in CM Fortran the assignment can also be an array operation. For example:

```
INTEGER, DIMENSION(5) :: A, B

DO I=1,5
   B(I) = 0          ! a front-end loop assignment
END DO

A = 0                ! a CM array assignment
```

Like any array operation, the array assignment that initializes **A** guarantees that **A** will be allocated on the CM. Array **B**, initialized with a serial loop, will be allocated on the front end *unless* it is used in an array operation elsewhere in the program unit. (Notice that if **B** is a CM array, the use of a loop to initialize it is much less efficient than an array operation would be.)

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

The values assigned to a CM array can be any expression of the appropriate type, including values retrieved by a **READ** statement or those generated by an array constructor (see Section 4.3.4).

Initialization is one of several uses of the specialized CM Fortran assignment statement **FORALL**, described later in this manual.

4.3.2 CMF_RANDOM Subroutine

CM Fortran provides a utility procedure for filling a numeric array with random values:

```
CMF_RANDOM( ARRAY, LIMIT )
```

The argument **ARRAY** must be an array object, residing on the CM. This procedure cannot be used with an array that is used only in the Fortran 77 manner (with scalar subscripts) in the program unit.

The range of values generated by **CMF_RANDOM** depends on the type of the argument array.

If **ARRAY** is of type integer, then the **LIMIT** argument serves as an exclusive upper bound. That is, the array is filled with random values in the range 0 to **LIMIT** − 1. If **ARRAY** is real or double-precision real, then **LIMIT** should be 1.0 and the values are in the range 0.0 to 1.0 (exclusive).

If **ARRAY** is real or double-precision real, the **LIMIT** value is ignored (and taken to be 1.0). In this case, the values returned are in the range 0.0 to 1.0 (exclusive). For example:

```
DIMENSION I(100), A(50,50)
CALL CMF_RANDOM( I, 1000 )     ! range: (0:999)
CALL CMF_RANDOM( A )           ! range: (0.0:1.0)
```

The utility library also provides a procedure specifically for 64-bit integers:

```
CMF_RANDOM_LONG_S_INTEGER( ARRAY, LIMIT )
```

This utility takes only an **INTEGER*8 ARRAY** argument, and it runs the cellular automaton that generates the values for 64 generations. The **LIMIT** argument is also an **INTEGER*8**. The other random number utility, **CMF_RANDOM**, accepts CM arrays of any numeric type, including **INTEGER*8**,

Copyright © 1989–1994 Thinking Machines Corporation

but it runs the automaton for only 32 generations, and it accepts only a 32-bit **LIMIT** argument.

To improve portability of the code you write, we recommend that you use the Fortran 90 intrinsic function **RANDOM_NUMBER** instead of **CMF_RANDOM** wherever possible. Note, however, that the argument to **RANDOM_NUMBER** must be of type real or double-precision real.

See the *CM Fortran Libraries Reference Manual* for more information about these and other utility procedures.

4.3.3 DATA Statement

A **DATA** statement can be used in the Fortran 77 fashion to initialize both scalars and arrays. It does not matter which syntax was used to declare the object.

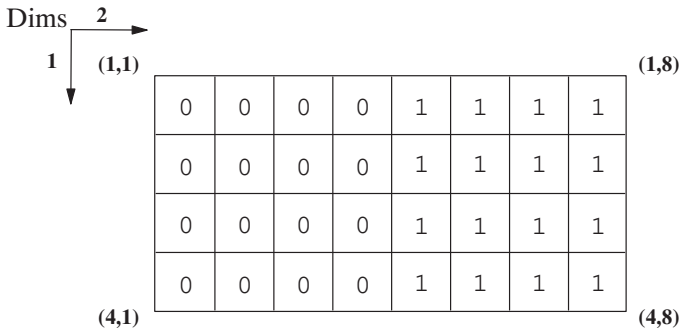
```
INTEGER I,A(5),B(10)
DATA I /1000/
DATA A / 1,2,3,4,5 /
DATA B / 10*0 /
```

As in Fortran 77, a **DATA** statement may also have an implied **DO** loop, which expands under the control of index variables to form a sequence of values. (The effect is similar to using a loop construct for dynamic initialization.) For example, given a 4 x 8 array **D**, the following two **DATA** statements initialize the left half to 0 and the right half to 1:

```
INTEGER, DIMENSION(4,8) :: D

DATA ( ( D(I,J), J=1,4 ), I = 1,4 ) / 16*0 /
DATA ( ( D(I,J), J=5,8 ), I = 1,4 ) / 16*1 /
```


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation



4.3.4 Array Constructors

An *array constructor* is a means of constructing an unnamed, 1-dimensional array by specifying an arbitrary sequence of scalar values.

Assigning an array constructor to a vector serves to initialize the vector with that sequence of values; the assignment is a CM operation, which causes the vector being assigned to be allocated on the CM. An array constructor may also be passed as an argument to a procedure, which has the same effect as passing a CM vector as an argument.

Array constructors can be specified in several ways. The simplest form is a list of the desired values, separated by commas and enclosed in square brackets:

```
INTEGER, DIMENSION(4) :: V

V = [ 10, 20, 30, 40 ]
```

If the values are not of the same type, all values are coerced to the type of the first value:

```
R = [ 10.0, 20, 30.0d0, 40 ] !R must be type REAL

C = [ (0.0,0.0), 1, 2.0, 3 ] !C must be type COMPLEX
```

Another form of array constructor uses a repeat count syntax, shown here with 100-element vectors:

Copyright © 1989–1994 Thinking Machines Corporation

```
A = [ 50[1], 50[0] ]          ! 50 1's and 50 0's

B = [ 25[ 1.0,2.0,3.0,4.0 ] ] ! sequence repeated 25 times
```

Array constructors can also take a form resembling the control variables of a **DO** loop or a triplet subscript:

[*first* : *last* : *increment*]

This form of array constructor uses only integer values. It builds a vector of values in the sequence specified; the resulting vector may be coerced to another type. For example,

```
C = [1:100]              ! integers from 1 to 100

D = REAL( [2:200:2] )    ! even reals from 2 to 200

E = CMPLX( [0:99],[0:99] )    ! sequence of 100 complex nos.

F = SQRT( REAL([1:500:5]) )  ! sequence of real square roots
```

4.3.5 DATA Attribute

The **DATA** attribute in a type declaration serves to initialize a scalar variable or an array.

The form for a scalar declaration is

type , **DATA** :: *name* = *value*

For example,

```
INTEGER, DATA :: SIZE = 1024
```

To declare and initialize an array, the declaration associates the **DATA** attribute with an array constructor. Since array constructors are always 1-dimensional, the **DATA** attribute can be applied only to 1-dimensional arrays. The form is

type , DIMENSION (*dim*), DATA :: *name* = [*array-constructor*]

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

For example,

```
INTEGER, DIMENSION(10), DATA :: J = [0,9,4,7,2,8,6,5,1,3]

REAL, DIMENSION(10), DATA :: B = [0.0,1:9]

DOUBLE PRECISION,DIMENSION(10),DATA::D=[5[0.0d0],5[1.0d0]]

COMPLEX, DIMENSION(10), DATA :: C = [ (0,0), 1:9 ]
```

As these examples suggest, the different forms of array constructor provide some flexibility in specifying sequences of values as the **DATA** attribute of an array. However, unlike a **DATA** statement, a **DATA** attribute cannot be associated with an implied **DO** loop.

When used in a type declaration, an array constructor is not considered an array operation and has no effect on the array's home. The arrays **J**, **R**, **B**, **D**, and **C** could be allocated either on the front end or on the CM, depending on how they are used in the program unit.

As shown in the case of **R**, the array constructor can be passed as an argument to a type-conversion function in the declaration. However, other elemental and transformational intrinsic functions — such as **RESHAPE** — cannot be used in specification statements.

4.4 A Note on Common Arrays

The features described in this chapter for declaration and dynamic initialization (assignment) apply to arrays in common blocks as well as to local arrays. However, the homes of common arrays are determined differently from those of local arrays, and the home can affect the method of *statically* initializing a common array. CM Fortran's treatment of common arrays is described in conjunction with subroutines in Chapter 6.

Copyright © 1989–1994 Thinking Machines Corporation