# Chapter 9

# More on Shapes and Parallel Variables

Chapter 3 introduced C* shapes and parallel variables. This chapter discusses more aspects of these important topics. Specifically:

- Partially specifying a shape; see Section 9.1.
- Creating copies of shapes; see Section 9.2.
- Dynamically allocating and deallocating a shape; see Sections 9.3 and 9.4.
- Using the C* library function **`palloc`** to explicitly allocate storage for a parallel variable; see Section 9.5.
- Casting to a shape, and casting to or from a parallel data type; see Section 9.6.

## 9.1 Partially Specifying a Shape

It is possible to declare a shape without fully specifying its rank and dimensions. You might do this, for example, if the number of positions in the shape is to be determined from user input. For example,

```
shape ShapeA;
```

declares a shape **`ShapeA`** but does not specify its rank or dimensions. Such a shape is *fully unspecified*.

```
shape []ShapeB;
```

specifies that **ShapeB** has a rank of 1, but does not specify the number of positions. Such a shape is *partially specified*.

You must fully specify a shape before using it (for example, before allocating parallel variables of that shape). Sections 9.2 and 9.3 describe ways of fully specifying a partially specified or fully unspecified shape.

The **rankof** intrinsic function returns 0 for a fully unspecified shape. For a partially specified shape, it returns the rank. For example, given these shapes:

```
shape s, [][]t, [8092]u;
```

These statements are true:

```
rankof(s) == 0;
rankof(t) == 2;
rankof(u) == 1;
```

This information can be used if you don't know whether or not a shape is fully specified — for example, in a function, where the function can fully specify a shape only if necessary.

### 9.1.1 Partially Specifying an Array of Shapes

You can also create an array of shapes that is partially specified. For example,

```
shape ShapeC[10];
```

declares that **ShapeC** is an array of 10 shapes, but does not specify the rank or dimensions of any of them.

```
shape [][]ShapeD[10];
```

declares that **ShapeD** is an array of 10 shapes, each of rank 2, but does not specify the number of positions in any of them.

A shape within such an array is specified with a right index in the standard manner. For example,

```
with (ShapeD[0])
```

makes the first shape in the array the current shape. Note that the shape must become fully specified before you can use it in this way.

You cannot use a parallel variable as an index into an array of shapes.

### Arrays and Pointers

The Standard C equivalence of arrays and pointers is maintained in C* with arrays of shapes and pointers to shapes. For example, if we declare a scalar pointer to **Sarray**:

```
shape *ptr;
ptr = Sarray;
```

then ***ptr** is equivalent to **Sarray[0]** and to ***Sarray**. Similarly,

```
Sarray[3]
```

is equivalent to

```
*(ptr + 3)
```

and to

```
*(Sarray + 3)
```

## 9.1.2 Limitations

You cannot partially specify the dimensions of a shape. This statement is incorrect:

```
shape [][4]ShapeE;      /* This is wrong */
```

Also, you cannot partially specify the rank of a shape. This statement is incorrect, if you later want to specify the shape as having a rank of 2:

```
shape []ShapeF;
```

A program cannot call the **positionsof** or **dimof** intrinsic function if the information the function requires has not yet been specified. If it is known when the program is being compiled that an error will result from such a call, the compiler reports an error. Otherwise, a run-time error is reported.

A shape must be fully specified before you can declare a parallel variable to be of that shape. You generally receive a compiler error if you try to declare a parallel variable to be of a shape that is not fully specified. A couple of exceptions:

- If the parallel variable is declared as an automatic in a nested scope. For example:

  ```
  shape ShapeA;

  main()
  {
      int:ShapeA p1;
  }
  ```

  In this case, the compiler assumes that **ShapeA** is fully specified elsewhere in the program. If it is not, a run-time error may be generated.

- If the shape has a storage class of **extern**. For example:

  ```
  extern shape ShapeB;
  int:ShapeB p2;
  ```

  In this case, the compiler assumes that **ShapeB** is fully specified in some other compilation unit, and a run-time error may be generated if it is not.

The next section describes how to, in effect, create copies of shapes. The section after that describes how to fully specify a partially specified or fully unspecified shape using the C* intrinsic function **allocate_shape**.

## 9.2 Creating Copies of Shapes

One way to fully specify a shape is by using the assignment operator to copy a fully specified shape to a partially specified one. For example:

```
shape ShapeA;
shape [256][256]ShapeB;
ShapeA = ShapeB;
```

In this case, both **ShapeA** and **ShapeB** refer to the same shape. You can use either one in a **with** statement to make this shape the current shape. This is different from what would happen if both were declared separately, but with the same dimensions. For example:

```
shape [256][256]ShapeA;
shape [256][256]ShapeB;
```

In this case, **ShapeA** and **ShapeB** refer to two separate physical shapes that happen to have the same rank and dimensions.

You can also fully specify a shape by using a shape-valued expression as the RHS of the assignment. For example:

```
ShapeA = shapeof(p1);        /* p1 is a parallel variable of
                                some other shape  */
ShapeB = (new_shape());      /* new_shape returns a shape */
ShapeC = *ptr;               /* ptr is a pointer to a shape */
```

### 9.2.1 Assigning a Local Shape to a Global Shape

Be careful when assigning a fully specified shape in local scope to a partially specified shape in file scope. This code illustrates the problem:

```
shape ShapeA;                /* Unspecified shape ShapeA */
void f(void)
{
    shape [1024][512]ShapeB; /* Fully specified shape ShapeB
                                in local scope */
    ShapeA = ShapeB;         /* ShapeB assigned to ShapeA */
}

main()
{
    f();
    {
         int:ShapeA p1;      /* This allocation fails because
                                ShapeA's shape was deallocated
                                when function f exited. */
    }
}
```

In this case, the actual physical shape that **ShapeA** refers to is allocated in local scope. When function **f** exits in the sample code, this shape is deallocated. When the code subsequently tries to declare a parallel variable of shape **ShapeA**, it gets an error, because the shape no longer exists.

The situation is analogous to what happens when a local pointer is assigned to a global pointer in Standard C.

## 9.3 Dynamically Allocating a Shape

Another way to fully specify a partially specified or fully unspecified shape is to use the C* intrinsic function **allocate_shape**. **allocate_shape**'s first argument is a pointer to a shape; its second argument is the rank of this shape; subsequent arguments are the number of positions in each rank. The function returns the shape it points to. For example,

```
shape []ShapeB;
ShapeB = allocate_shape(&ShapeB, 1, 65536);
```

completes the specification of the partially specified 1-dimensional shape **ShapeB**.

You needn't partially specify a shape before calling **allocate_shape**. For example,

```
allocate_shape(&new_shape, 3, 2, 2, 4096);
```

returns a 3-dimensional shape called **new_shape**.

**allocate_shape** can also fully specify elements of an array of shapes. For example:

```
ShapeD[0] = allocate_shape(&ShapeD[0], 2, 4, 16384);
```

Alternatively, you can use an array to specify the number of positions in each rank. This format is useful if the program will not know the rank until run time, and therefore can't use the variable number of arguments required by the previous syntax. The example below reads the rank and dimensions in from a file named **shape_info** and uses these values as arguments to **allocate_shape**.

```
#define  MAX_AXES  31
#include <stdio.h>

main()
{
    FILE *f;
    int axes[MAX_AXES], i, rank;
    shape ShapeA;

    f = fopen("shape_info", "r");

    fscanf(f, "%d", &rank);
    if (rank > MAX_AXES) {
         fprintf (stderr, "Rank bigger than maximum
```

```
                    allowed.\n");
          exit(1);
     }
     for (i = 0; i < rank; i++)
          fscanf(f, "%d", &axes[i]);

     ShapeA = allocate_shape(&ShapeA, rank, axes);
}
```

Note that **axes** is initialized as an array of 31 elements, since the CM restricts shapes to a maximum of 31 dimensions. Of course, the file **shape_info** could contain fewer than the maximum number of dimensions.

NOTE: For certain programs you may be able to improve performance by using the intrinsic function **allocate_detailed_shape** instead of **allocate_shape**. Appendix A discusses this function for CM-200 C*; Appendix B discusses it for CM-5 C*.

## 9.4 Deallocating a Shape

Use the C* library function **deallocate_shape** to deallocate a shape that was allocated using the **allocate_shape** function. Its argument is a pointer to a shape. Include the header file **<stdlib.h>** if you call **deallocate_shape**. Note that this is not required for **allocate_shape**, which is an intrinsic function.

There are two reasons to deallocate a shape:

- If you have reached the limit on the number of shapes imposed by your CM system. To avoid this, in general you should deallocate a shape when you leave the scope in which the shape is defined.
- If you want to reuse a partially specified shape.

As an example of the latter, consider this code:

```
#include <stdlib.h>

shape []S;
int positions = 4096;

main()
```

```
{
    while (positions<=65536) {
         S = allocate_shape(&S, 1, positions);
              {
                   int:S p1, p2, p3;
                   /* Parallel code omitted ... */
              }
         deallocate_shape(&S);
         positions *= 2;
    }
}
```

In this code, shape **S** is allocated every time it goes through the **while** loop, and deallocated at the end of the loop. This lets it have a different number of positions each time through the loop.

The results of deallocating a shape that was fully specified at compile time are undefined.

You should not deallocate a shape when there are parallel variables of that shape still allocated; if you do, the behavior of these parallel variables is undefined. Note that in the code fragment above, the parallel variables declared to be of shape **S** go away when you leave the block.

As discussed in Section 9.2, you can create copies of shapes by assigning one shape to another. If you have created copies of shapes in this way and you deallocate one, the effect on the others is undefined.

## 9.5 Dynamically Allocating a Parallel Variable

The C* library routine **palloc** is the parallel equivalent of C library routines like **malloc** and **calloc**. Use it to explicitly allocate storage for a parallel variable. It can be called whether or not the parallel variable's shape is dynamically allocated. Include the file **<stdlib.h>** if you call **palloc** or its companion function **pfree**.

**palloc** takes two arguments: a shape, and a size (in **bool**s). It allocates space of that size and shape, and returns a scalar pointer to the beginning of the allocated space. The shape passed as an argument must be fully specified before **palloc** is called.

**palloc** returns 0 if it cannot allocate the memory.

To allocate space for a parallel variable of shape **ShapeA**, for example, you could do this:

```
#include <stdlib.h>

shape [16384]ShapeA;
int:ShapeA *ptr;

main()
{
    ptr = palloc(ShapeA, boolsizeof(int:ShapeA));
}
```

The scalar variable **ptr** now contains a pointer to an **int**-sized parallel variable of shape **ShapeA**. You can reference this parallel variable by using ***ptr**. The contents of the parallel variable are undefined.

Use **pfree** to deallocate storage you allocated with **palloc**. **pfree** takes as its argument the pointer returned by **palloc**. For example, to deallocate the storage allocated by the call to **palloc** above, call **pfree** as follows:

```
pfree(ptr);
```

The **palloc** and **pfree** calls can also be used with a dynamically allocated shape, as in this example:

```
#include <stdlib.h>

shape S;
double:S *p;

main()
{
    S = allocate_shape(&S, 2, 4, 8192);
    p = palloc(S, boolsizeof(double:S));
    /* ... */
    pfree(p);
    deallocate_shape(&S);
}
```

Note that you are responsible for freeing the storage you allocate before you free the associated shape.

Also, note that you can declare a scalar pointer to a parallel variable of a shape that is not fully specified, even though you cannot declare a parallel variable of that shape.

## 9.6 Casting with Shapes and Parallel Variables

Use the C* cast operator to cast an expression to a particular shape and type. For example,

```
(char:employees)
```

specifies that the expression following it is to be formed into a **char** of shape **employees**. You must specify a data type as well as a shape in a parallel cast; there are no defaults.

### 9.6.1 Scalar-to-Parallel Casts

Using a parallel cast is a quick way to promote a scalar value. The statement below stores in scalar variable **s1** the number of active positions of the current shape:

```
s1 = +=(int:current)1;
```

In the statement, 1 is cast to a parallel **int** of the current shape. The **+=** reduction operator sums the resulting parallel variable for all active positions, and the result is assigned to the scalar variable **s1**.

### 9.6.2 Parallel-to-Parallel Casts

Parallel-to-parallel casts are also permitted.

#### Casts to a Different Type

You can cast a parallel variable so that it has a different type. For example:

```
int:ShapeA p1;
sqrt((double:ShapeA)p1);
```

The parallel version of **`sqrt`** requires a **`float`** or a **`double`**; therefore, we must cast the parallel **`int p1`** before we can pass it to this function.

#### Casts to a Different Shape

Casting of a parallel variable to a different shape is limited to the situation in which the same shape can be referenced by more than one name. In this case, a cast may sometimes be necessary to ensure that the compiler recognizes that two parallel variables are supposed to be of the same shape. For example:

```
shape [256][256]ShapeB, ShapeA;

main()
{
  ShapeA = ShapeB;
  {
   int a:ShapeA, b:ShapeB;
    with(ShapeB) {
         b = a;    /* This gets a compile-time error */
         b = (int:ShapeB)a;  /* This works */
         }
    }
}
```

The cast is required so that the compiler is made aware that **`ShapeA`** and **`ShapeB`** refer to the same shape.

No movement of data is implied in a parallel-to-parallel cast.

The effects of casting an expression between two shapes that are different (for example, with a different rank or number of positions) are undefined.

### 9.6.3 With a Shape-Valued Expression

You can use a shape-valued expression with a scalar-to-parallel or parallel-to-parallel cast. The expression must be enclosed in parentheses unless it is an intrinsic function. For example,

```
s1 = +=(int:(shape_array[3]))1;
```

casts 1 to be an **int** of the fourth shape in the array **shape_array**.

### 9.6.4 Parallel-to-Scalar Casts

You can cast a parallel variable to a scalar type. The result is similar to a demotion of a parallel variable when assigning it to a scalar (see Chapter 5); the operation picks one of the active values of the parallel variable and returns that as the result. If no positions are active, the result of the cast is undefined.

## 9.7 Declaring a Parallel Variable with a Shape-Valued Expression

A shape-valued expression, as we have described earlier, is an expression that can be used in place of a shape name. You can therefore use a shape-valued expression in declaring a parallel variable. The expression must be enclosed in parentheses unless it is the **shapeof** intrinsic function. For example:

```
shape [256][256]matrix;
int:matrix p1;
int:shapeof(p1) p2;      /* p2 is of shape matrix */
int:(get_shape()) p3;    /* get_shape returns a shape */
```

However, if the declaration appears at file scope, or is **static** or **extern**, the shape-valued expression must be a constant. This means that the expression must be one of the following:

- A simple shape that is fully specified at compile time, or that has a storage class of **extern**. For example, **shapeof** in the example above refers to a fully specified shape.
- An array of shapes that is fully specified at compile time and whose right index is a constant expression. For example:

  ```
  shape [256][512]Sarray[40];
  int:(Sarray[17]) p1;
  int:(Sarray[4-3]) p2;
  ```

- An indirection of an array of shapes that is fully specified at compile time, with a constant expression added to it. For example:

```
shape [512][256]Sarray[40];
int:(*(Sarray + 17)) p1;
int:(*(Sarray + 4 - 3)) p2;
```

These are illegal:

```
shape Sarray1[40];
int:(Sarray1[17]) p1;    /* This is wrong */
```

**Sarray1** is not fully specified; therefore, you can't declare **p1** to be a parallel variable of any of the elements of it.

```
shape [512][256]Sarray[40];
int:(Sarray[f(x)]) p1;    /* This is wrong */
```

In this case, **Sarray** is fully specified, but **f(x)** is not a constant expression, since it invokes a function whose result is not known until run time.

```
shape *ptr;
int:(*ptr) p1;     /* This is wrong */
```

In this case, **ptr** does not point to a fully specified shape.

## 9.8 The physical Shape

C* contains the predeclared shape name **physical**; **physical** is a new keyword that C* adds to Standard C. The shape **physical** is always of rank 1; its number of positions is the number of physical processors on which your program is running. (In the CM-5 implementation, it is either the number of nodes or the number of vector units, depending on how you compiled the program. See the *CM-5 C* User's Guide* for more information.) Note, therefore, that the number of positions in the shape is not known until run time.

You can use **physical** as you would any other shape. For example,

```
positionsof(physical);
```

returns the number of positions in shape **physical**, which is equal to the number of physical processors on which the program is running.

```
(int:physical)p1
```

casts **p1** to be an **int** of shape **physical**.