Chapter 3

Shapes and Parallel Variables

As the sample program in Chapter 2 shows, shapes and parallel variables are central to the way C* extends C to support data parallel programming. This chapter describes shapes and parallel variables in more detail.

3.1 Shapes

A shape is a template for parallel data. In C*, you must specify the shape of the data before you can define data of that shape. A shape is specified by:

- How many dimensions it has. This is referred to as the shape's *rank*.
- The number of *positions* in each of its dimensions. A position is an area that can contain values of parallel data.

The total number of positions in a shape is the product of the number of positions in each of its dimensions. For example, an 8-by-4 shape has 32 positions.*

Typically, the choice of a shape reflects the natural organization of the data. For example, a graphics program might use a shape representing the 2-dimensional images that the program is to process. If your program works on different data sets, you can have a different shape for each one.

* In the CM-200 implementation of C*, the number of positions in each dimension of a shape must be a power of 2, and the total number of positions in the shape must be some multiple of the number of physical processors in the section of the CM that the C* program is using.

Declaring Shapes

Use the new C* keyword **shape** to declare a shape, as in this example:

```
shape [16384]employees;
```

This statement declares a shape called **employees**. It has one dimension (a rank of 1) and 16384 positions. A dimension is also referred to as an *axis*.

Note the position of the brackets, to the left of the shape name. *Left indexing* is an important new concept in C*. A shape can have multiple axes; specify each in brackets to the left of the shape name. For example, here is a 2-dimensional shape:

```
shape [256][512]image;
```

The left-most axis is referred to as axis 0; the next axis to the right is axis 1, and so on.

A program can include many shapes. You can use a single shape statement to declare multiple shapes. For example:

```
shape [16384]employees, [256][512]image;
```

The shapes we have looked at so far have been *fully specified*. You don't need to specify a shape fully when you declare it. For example,

```
shape data;
```

declares a shape called **data**, without specifying its rank or the number of positions in it. You can also specify the shape's rank without specifying its size. For example,

```
shape []data;
```

declares a 1-dimensional shape of unspecified size.

Declaring a shape that is not fully specified is useful if, for example, the size of the shape is to come from user input. However, a shape's rank and size must be fully specified before you can use it. Section 3.7 describes how to specify a shape fully.

3.2 Parallel Variables

As Chapter 2 explained, a parallel variable is similar to a Standard C variable, except that it has a shape in addition to its type and storage class. The shape defines how many *elements* of a parallel variable exist, and how they are organized. Each element occupies one position within the shape and contains a single value. If a shape has 16384 positions, a parallel variable of that shape has 16384 elements, one for each position.

Each element of a parallel variable can be thought of as a single scalar variable. But a C* program can also carry out operations on all elements (or any subset of elements) of a parallel variable at the same time.

Declaring Parallel Variables

Before declaring a parallel variable, you must fully specify the shape that the parallel variable is to take. For example, once you have declared shape **`employees`** as in the example above, you can declare a parallel variable of that shape:

```
unsigned int:employees employee_id;
```

This statement declares a parallel variable named **`employee_id`**; it is an **`unsigned int`** of shape **`employees`**. Figure 7 shows this parallel variable.

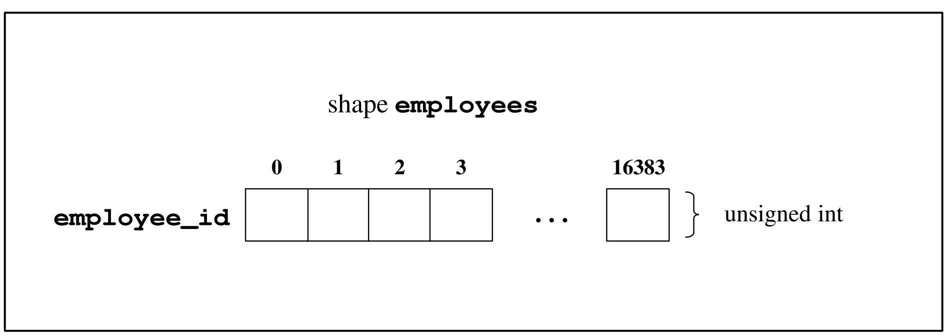

Figure 7. A parallel variable of shape `employees`.

Once the parallel variable has been declared, you can use left indexing to specify an individual element of it. For example, **`[2]employee_id`** refers to the third element of **employee_id** in Figure 7. **[2]** is referred to as the *coordinate* for this element.

You can declare many parallel variables of the same shape. If they are of the same type, you can declare them in the same statement. For example:

```
unsigned int:employees employee_id, age, salary;
```

All three parallel variables are of shape **employees**. As shown in Figure 8, elements of different parallel variables that are in the same position of a shape are referred to as *corresponding elements*.

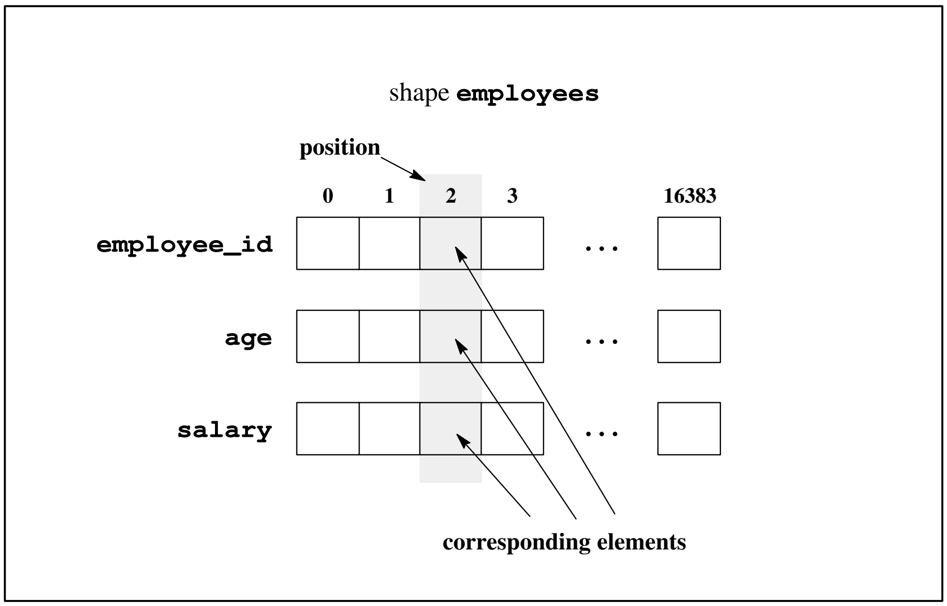

Figure 8. Three parallel variables of shape employees.

3.3 Other Kinds of Parallel Data

In addition to parallel variables, C* provides parallel versions of C aggregate types. In this section, we look at parallel structures and parallel arrays.

Parallel Structures

You can declare an entire structure as a parallel variable. For example, if you have declared this shape and structure:

```
shape [16384]employees;
struct date {
   int month;
   int day;
   int year;
   };
```

you can declare a parallel structure as follows:

```
struct date:employees birthday;
```

`birthday` is of type **`struct date`** and of shape **`employees`**. It is shown in Figure 9.

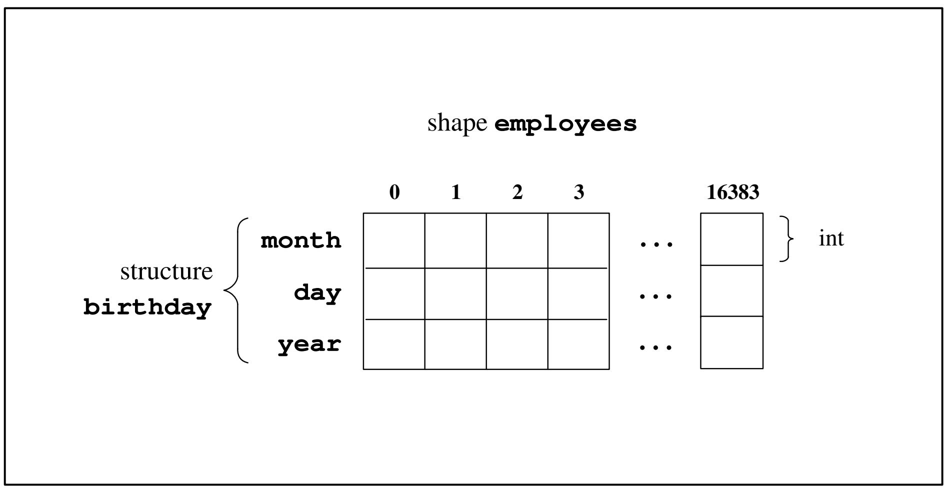

Figure 9. **A parallel structure of shape `employees`.**

Each element of the parallel structure contains a scalar structure, which in turn will contain the birthday of an employee.

Accessing a member of a parallel structure works the same way as accessing a member of a scalar structure. For example, **`birthday.day`** specifies all elements of structure member **`day`** in the parallel structure **`birthday`**.

C* does not allow shapes, pointers, or parallel variables inside a parallel structure.

Parallel Arrays

You can also declare an array of parallel variables. For example,

```
shape [16384]employees;
int:employees ratings[3];
```

declares an array of three parallel variables of shape **employees**, as shown in Figure 10.

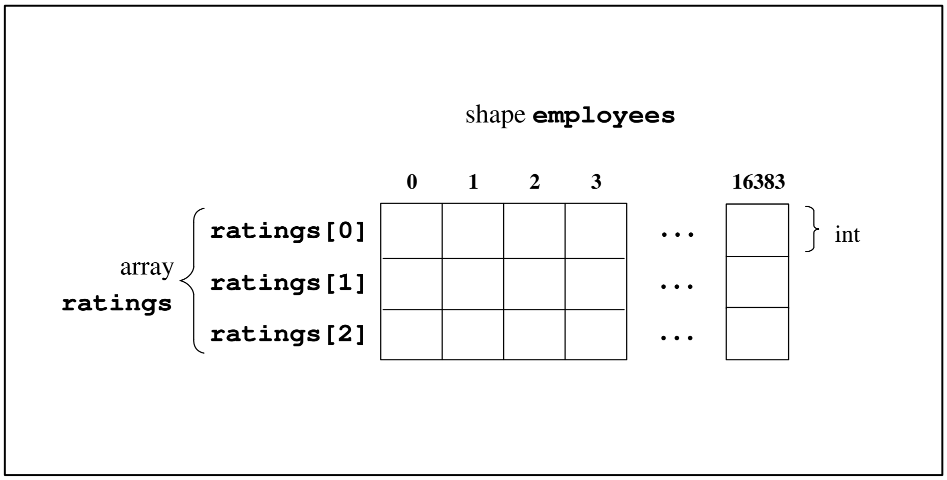

Figure 10. A parallel array of shape employees.

3.4 Pointers to Shapes and Parallel Variables

In addition to Standard C pointers, C* provides pointers to shapes and parallel variables. As with C pointers, you can use the pointer in place of the object being pointed to. In functions, you can use pointers to pass by reference instead of by value.

Pointers to Shapes

This statement declares the scalar variable **`shape_ptr`** to be a pointer to a shape:

```
shape *shape_ptr;
```

And this statement makes **`shape_ptr`** point to shape **`ShapeA`**:

```
shape_ptr = &ShapeA;
```

A dereferenced pointer to a shape can be used as a *shape-valued expression*; that is, you can use it wherever you would use a shape name. For example,

```
with (*shape_ptr)
```

makes **`ShapeA`** the current shape.

Pointers to Parallel Variables

The statement:

```
int:ShapeA *pvar_ptr;
```

declares a scalar pointer **`pvar_ptr`** that points to a parallel **`int`** of shape **`ShapeA`**. If **`p1`** is a parallel variable of shape **`ShapeA`**, then

```
pvar_ptr = &p1;
```

makes **`pvar_ptr`** a pointer to **`p1`**. You can then reference **`p1`** via the pointer **`pvar_ptr`**.

You can declare a pointer to a parallel variable of a shape that is not fully specified, even though you cannot declare a parallel variable of that shape. For example:

```
shape data;
int:data *data_ptr;
```

The relationship between arrays and pointers is maintained in C*. For example,

```
int:ShapeA A1[40];
```

declares a parallel array of 40 **ints** of shape **ShapeA**, and **A1** points to the first element of the array — that is, its type is a scalar pointer to a parallel **int** of shape **ShapeA**.

3.5 Choosing a Shape: The with Statement

Before you can carry out most operations on parallel variables, they must be of the *current shape*. You designate the current shape by using the new C* **with** statement. For example:

```
shape [16384]employees;
unsigned int:employees employee_id, age, salary;

main()
{
   with (employees)
      /* Operations on parallel variables of shape
         employees go here. */
}
```

Most operations on parallel variables of shape **employees** can occur only within the scope of the **with** statement. We will discuss what constitutes an operation on a parallel variable in the next chapter. For now, note that the **with** statement does not restrict operations on scalar expressions — that is, expressions, like Standard C expressions, that refer to a single data point. This includes parallel variables that are left-indexed so that they specify only one element of the parallel variable — for example, **[4]age** is considered a scalar expression. Dereferenced pointers to parallel variables, however, share the same restrictions as parallel variables.

You can have many **with** statements in a program, making different shapes current at different times. You can also nest **with** statements. When the program returns from the nested **with** statement, the previous shape once again becomes current, as in this example:

```
with (ShapeA) {
   /* Put operations on parallel variables of shape
      ShapeA here. */
```

```
      with (ShapeB) {
         /* Operations on variables of shape ShapeB. */
      }
      /* Operations on variables of shape ShapeA
         once again. */
   }
```

3.6 Setting the Context: The where Statement

To perform an operation on a subset of the elements of a parallel variable, use the new C* **where** statement to restrict the context in which the operation is performed. A **where** statement specifies which positions in a shape remain *active*; code in the body of a **where** statement operates only on elements in active positions.

For example, the code below restricts parallel operations to positions of shape **employees** where the value of parallel variable **age** is greater than 35:

```
shape [16384]employees;
unsigned int:employees employee_id, age, salary;

main()
{
   with (employees)
      where (age > 35)
         /* Parallel code in restricted context
            goes here. */
}
```

The controlling expression that **where** evaluates to set the context must operate on a parallel variable of the current shape. It evaluates to 0 (false) or non-zero (true) separately for each position that is currently active. Positions in which the expression is false are made inactive. If no positions are active, code is still executed, but an operation on a parallel variable of the current shape has no result. Initially, all positions in all shapes are active.

You can nest **where** statements; the effect is to cumulatively shrink the set of active positions of the current shape. For example, the two **where** statements below restrict the context to positions of shape **employees** in which **age** is greater than 35 and **salary** is greater than 50000:

```
with (employees)
   where (age > 35) {
      /* Parallel code in restricted context
         goes here. */
      where (salary > 50000) {
         /* Parallel code in more restricted context
            goes here. */
      }
   }
```

Like the **if** statement in Standard C, the **where** statement can include an **else** clause. The **else** clause reverses the set of active positions; that is, those positions that were active when the **where** statement was executed are made inactive, and those that were made inactive are made active. In the example below, it causes parallel operations to be carried out on positions of shape **employees** where **age** is less than or equal to 35 (assuming that all positions are initially active):

```
with (employees)
   where (age > 35)
      /* Parallel code in one restricted context. */
   else
      /* Parallel code in the opposite context. */
```

The everywhere Statement

C* also provides an **everywhere** statement. The **everywhere** statement makes all positions of the current shape active. Parallel code within the scope of an **everywhere** statement operates on all positions of the current shape, no matter what context has been set by previous **where** statements. After the **everywhere** statement, the context returns to what it was before the **everywhere**.

3.7 Dynamically Allocating Shapes and Parallel Variables

One of the powerful features of C* is that it allows you to allocate (and deallocate) shapes and parallel variables dynamically. The functions

allocate_shape and **deallocate_shape** do this for shapes; **palloc** and **pfree** do it for parallel variables.

The allocate_shape and deallocate_shape Functions

Use the C* intrinsic function **allocate_shape** to allocate a shape dynamically. The **allocate_shape** function has two formats. In the first, it takes as arguments a pointer to a shape, the rank of this shape, and the number of positions in each axis. It returns a description of the shape. For example,

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

allocates a 3-dimensional shape that is 2-by-2-by-4096.

In the alternative format, you can use an array to specify the rank and the number of positions in each axis. 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.

You can use **allocate_shape** either to allocate a totally new shape, or to complete the specification of a shape that was not fully specified when you declared it.

Use the C* library function **deallocate_shape** to deallocate a shape that was allocated via **allocate_shape**; include the header file **<stdlib.h>** when calling this function. Its argument is a pointer to the shape to be deallocated. For example,

```
deallocate_shape(&new_shape);
```

deallocates the shape allocated above.

You might want to deallocate a shape if you have reached the limit on the number of shapes imposed by your CM system, or if you want to reuse a partially specified shape.

For certain programs, you may be able to improve performance by using the intrinsic function **allocate_detailed_shape**. See the *C* Programming Guide* for information.

The palloc and pfree Functions

Use the C* library function **palloc** to explicitly allocate storage for a parallel variable; use the function **pfree** to free this storage. In both cases, include the header file **<stdlib.h>**.

The **palloc** function takes two arguments: a shape and a size. It allocates space of that size and shape, and returns a pointer to the beginning of the space.

The unit of size in **palloc** is **bool**s. **bool** is a new data type in C* that stores either a 0 or a 1. (The actual size of a **bool** can be different in different implementations.)

To obtain the exact size of a variable or data type in units of **bool**s, use the new C* operator **boolsizeof**. For example,

```
s1 = boolsizeof(int:ShapeA);
```

returns the number of **bool**s that must be allocated for a single instance of a parallel **int**.

The **pfree** function takes as its argument the pointer returned by **palloc**.

The example below allocates a shape and a parallel variable of that shape, using **allocate_shape** and **palloc**, then deallocates both using **pfree** and **deallocate_shape**:

```
#include <stdlib.h>

shape S;
double:S *p;

main()
{
   allocate_shape(&S, 2, 4, 8192);
   p = palloc(S, boolsizeof(double:S));

   /* ... */

   pfree(p);
   deallocate_shape(&S);
}
```

Note the use of **boolsizeof** to obtain the size, in **bool**s, of a parallel **double**.