Chapter 3

Using Shapes and Parallel Variables

The sample C* program in Chapter 2 began by declaring a shape and several parallel variables. Shapes and parallel variables are the two most important additions of C* to Standard C. This chapter introduces these topics; Chapter 9 discusses them in more detail.

3.1 What Is a Shape?

A shape is a template for parallel data, a way of logically configuring data. In C*, you must define the shape of the data before you can operate on it. A shape is defined by:

- The number of its dimensions. This is referred to as the shape's *rank*. For example, a shape of rank 2 has two dimensions. A shape can have from 1 to 31 dimensions. A dimension is also referred to as an *axis*.
- The number of *positions* in each of its dimensions. A position is an area that can contain individual 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. Thus, a 2-dimensional shape with 4 positions in axis 0 (the first dimension) and 8 positions in axis 1 (the second dimension) has 32 total positions, organized as shown in Figure 7. (By convention in this guide, axis 0 denotes the row number, and axis 1 denotes the column number.)

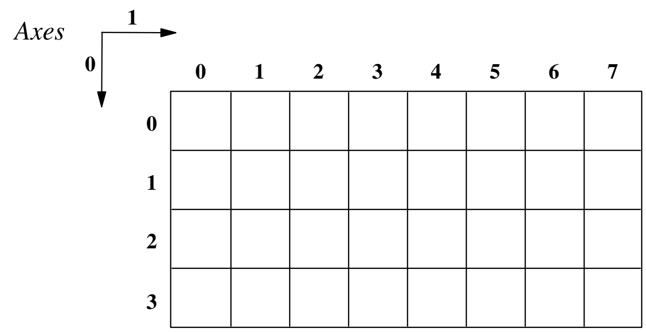

Figure 7. A 4-by-8 shape.

NOTE FOR USERS OF CM-200 C*

The CM-200 implementation of C* imposes these restrictions on shapes in C*:

- The number of positions in each dimension of a shape must be a power of two.
- 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.

For example, if the program is running in a CM section with 8192 physical processors, it can have shapes with 8192 positions, 16384 positions, and so on. You can arrange them 2 by 4096, 4 by 4 by 512, and so on.

3.2 Choosing a Shape

The choice of a shape depends on the data that the C* program is going to be using. The shape typically reflects the natural organization of the data. For example:

- A database program for the employee records of a large company might use a 1-dimensional shape, with the number of positions equaling the number of employees.
- A graphics program might use a shape representing the 2-dimensional images that the program is to process. If the images have 256 pixels in the vertical dimension and 256 pixels in the horizontal dimension, a shape of rank 2 with 256 positions in each dimension would be appropriate. This would let each position represent a pixel in an image.
- A program to analyze stress in a solid object might use a 3-dimensional shape, with each axis representing a dimension of the object, and each position representing some portion of the volume of the object.

3.3 Declaring a Shape

Here is a declaration of a shape in C*:

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

This statement declares a shape called **`employees`**. It has one dimension (a rank of 1) and 16384 positions.

Let's take a closer look at the components of the statement:

- **`shape`** is a new keyword that C* adds to Standard C.
- **`[16384]`** specifies the number of positions in the shape. If the shape is declared at file scope, or as an **`extern`** or **`static`** at block scope, the value in brackets must be a constant expression. Otherwise, it can be any expression that can be evaluated to an integer. This follows the ANSI C standard.

- **`employees`** is the name of the shape. Shape names follow standard C naming rules. They are in the same name space as variables, functions, **`typedef`** names, and enumeration constants.

Figure 8 shows the shape declared above.

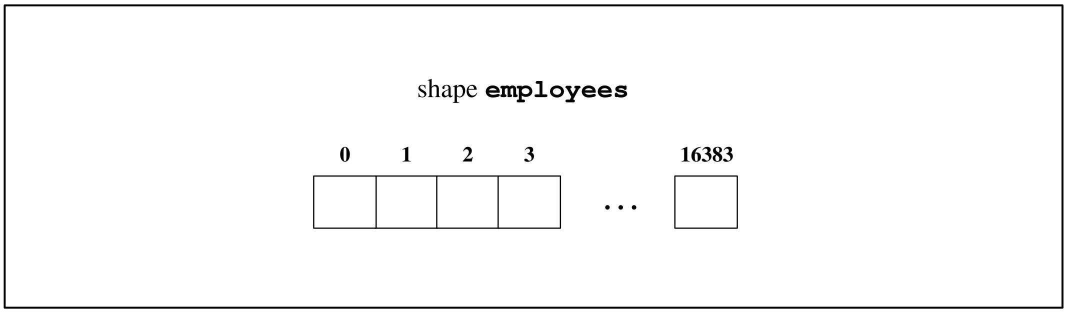

Figure 8. The shape `employees`.

A 2-dimensional shape adds another number, in brackets, to the right of the first set of brackets. This number represents the number of positions in the second dimension. For example:

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

This shape has 256 positions along axis 0 and 512 positions along axis 1. Each additional dimension is represented by another number in brackets, to the right of the previous dimensions.

Individual positions within a shape can be identified using bracketed numbers as *coordinates*. For example, position [4] of shape **`employees`** is the fifth position in the shape (numbering starts with 0, as in C). Position [47][112] of shape **`image`** is the position at coordinate 47 along axis 0 and 112 along axis 1.

3.3.1 Declaring More Than One Shape

A program can include many shapes. You can use a single **`shape`** statement to declare more than one shape. For example:

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

3.3.2 The Scope of a Shape

A shape's scope is the same as that of any identifier in Standard C. For example, a shape declared within a function or block is local to that function or block. A shape declared at global scope can be referenced anywhere in the source file after its declaration.

NOTE: If a block contains a shape declaration, you should not branch into it (for example, with a **switch** or **goto** statement); the behavior is undefined.

3.4 Obtaining Information about a Shape

You can obtain information about a shape by using the C* intrinsic functions **positionsof**, **rankof**, and **dimof**. (Intrinsic functions are new in C*; they have function-like syntax, but they must be known to the compiler — for example, because they don't follow all Standard C rules for functions.)

- **positionsof** takes a shape as an argument and returns the total number of positions in the shape.
- **rankof** takes a shape as an argument and returns the shape's rank.
- **dimof** takes two arguments: a shape and an axis number. It returns the number of positions along that axis.

The simple C* program below displays information about a shape.

```
#include <stdio.h>

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

main()
{
 printf ("Shape 'employees' has rank %d and %d positions.\n",
            rankof(employees), positionsof(employees));
 printf ("Shape 'image' has rank %d and %d positions.\n",
            rankof(image), positionsof(image));
 printf ("Axis 0 has %d positions; axis 1 has %d positions.\n",
            dimof(image,0), dimof(image,1));
}
```

Its output is:

```
Shape 'employees' has rank 1 and 16384 positions.
Shape 'image' has rank 2 and 131072 positions.
Axis 0 has 256 positions; axis 1 has 512 positions.
```

These intrinsic functions can be used in other, more interesting contexts, as we discuss later.

3.5 More about Shapes

So far, we have covered the basics about shapes in C*. Chapter 9 discusses more advanced aspects of shapes. For example:

- Partially specifying a shape
- Copying shapes
- Dynamically allocating a shape

3.6 What Is a Parallel Variable?

Once a program has declared a shape, it can declare variables of that shape. These variables are called *parallel variables*.

3.6.1 Parallel and Scalar Variables

A good way to understand parallel variables is to compare them with standard C variables. As we mentioned in Chapter 2, Standard C variables are referred to in this guide as *scalar* to distinguish them from parallel variables. A scalar variable contains only one "item" — one number, one character, and so on. A parallel variable contains many items. (Note that Standard C uses the term *scalar* in a slightly different way, to refer collectively to arithmetic and pointer types. We consider a Standard C array or structure, for example, to be scalar because it contains only one array or structure.)

A scalar variable has the following associated with it:

- a type, along with its modifiers and qualifiers, (for example, **`char`**, **`unsigned int`**, **`long double`**) that defines how much memory is to be allocated for the variable and how operators deal with it

- a storage class (for example, **`auto`**, **`static`**) that defines the manner in which the memory is to be allocated

Like a scalar variable, a parallel variable has a type and a storage class, but in addition it has a *shape*. The shape defines how many *elements* of a parallel variable exist, and how they are organized. Each element exists in a position in the shape and contains a single value for the parallel variable. 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 the advantage of a parallel variable is that C* lets a program carry out operations on all elements (or any subset of elements) of a parallel variable at the same time. As the sample program in Chapter 2 demonstrated, you can:

- Assign a constant to all elements of a parallel variable at the same time.

- Declare multiple parallel variables of the same shape.

- Perform an arithmetic operation on all elements of a parallel variable at the same time.

- Do reduction assignments of data in all elements of a parallel variable.

As we explain later in this manual, parallel variables that have different shapes can interact, but interactions between parallel variables are more efficient if the parallel variables are of the same shape.

3.7 Declaring a Parallel Variable

Before declaring a parallel variable, you must define the shape that the parallel variable is to take. For example, assume that this shape has been defined:

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

You can then declare parallel variables of this shape. For example:

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

Interpret the colon in this syntax to mean "of shape *shapename*." Thus, this statement declares a parallel variable called **`employee_id`** that is of shape **`employees`**. **`unsigned int`** specifies the type of the parallel variable **`employee_id`**. Parallel variable names, like shape names, follow Standard C naming rules.

Figure 9 shows this parallel variable.

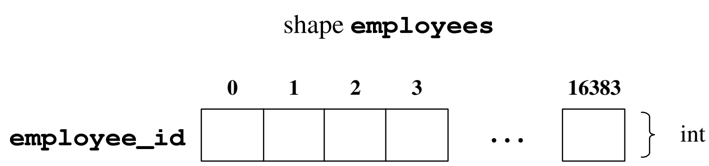

Figure 9. A parallel variable of shape `employees`.

3.7.1 Declaring More Than One Parallel Variable

You can declare more than one parallel variable in the same statement, if they are of the same type. For example:

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

The parallel variables need not be of the same shape. For example:

```
unsigned int employee_id:employees, field1:image;
```

A Shortcut for Declaring More Than One Parallel Variable

If parallel variables have the same type and same shape, C* provides a more concise method for declaring them. Put the "`:`*shapename*" after the type rather than after each parallel variable. For example:

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

The parallel variables **`employee_id`**, **`age`**, and **`salary`** are all **`unsigned ints`** of shape **`employees`**. This syntax is generally used except when parallel variables of different shapes are being declared.

Figure 10 shows the three parallel variables that this statement creates.

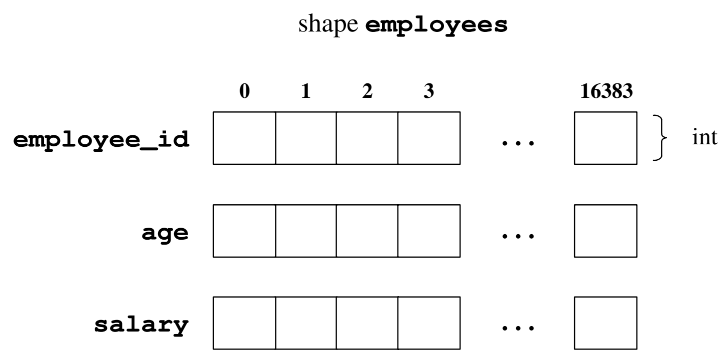

Figure 10. Three parallel variables of shape `employees`.

3.7.2 Positions and Elements

As we have mentioned, a shape is a template for the creation of parallel variables. It is important to keep in mind the distinction between positions of a shape and elements of parallel variables that have been declared to be of that shape. As shown in Figure 11, elements with the same coordinates can be considered to occupy the same position in the shape. For example, the third elements of **`em-ployee-id`**, **`age`**, and **`salary`** are all at position [2] of shape **`employees`**. These elements are referred to as *corresponding elements*. Corresponding elements are an important concept in C*.

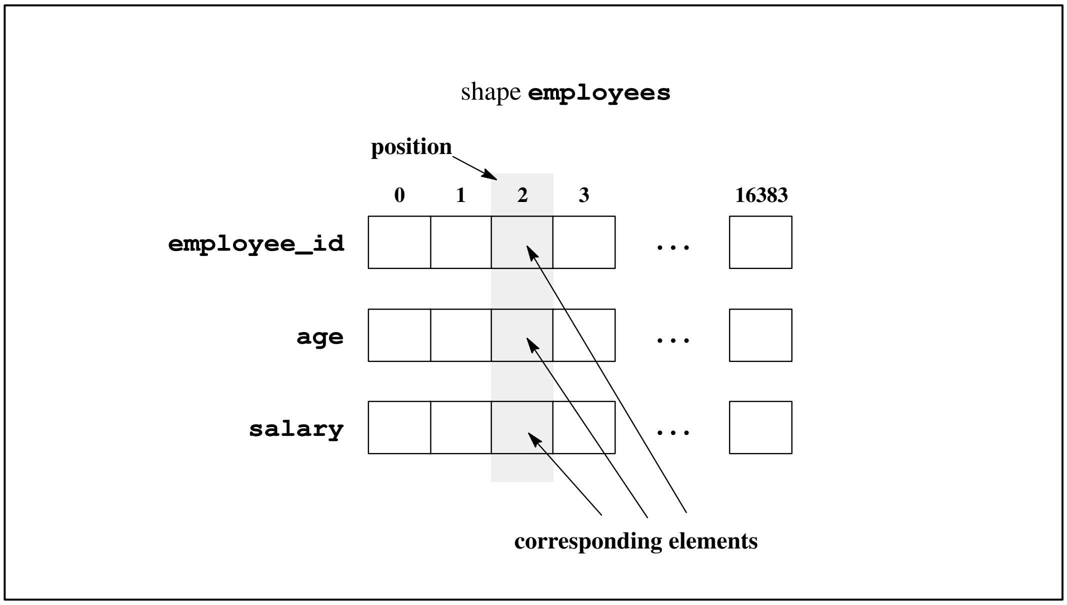

Figure 11. Corresponding elements.

3.7.3 The Scope of Parallel Variables

Parallel variables follow the same scoping rules as standard scalar variables (and shapes). For example, a parallel variable declared within a function or block is local to that function or block.

NOTE: As with shape declarations, if a block contains a parallel variable declaration, you should not branch into it (for example, with a **switch** or **goto** statement); the behavior is undefined.

3.8 Declaring a Parallel Structure

You can declare an entire structure as a parallel variable. For example:

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

```
        int year;
        };
    struct date:employees birthday;
```

The final line of code defines a parallel variable called **birthday**. It is of shape **employees** and of type **struct date**. This parallel structure is shown in Figure 12.

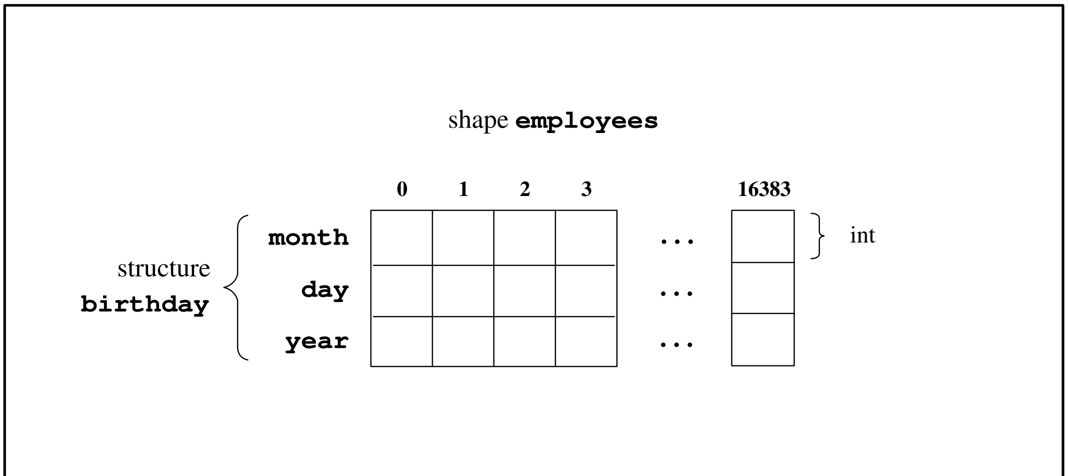

Figure 12. A parallel structure of shape `employees`.

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

As with non-structured variables, you can declare more than one parallel structure in a single statement. For example:

```
struct date:employees birthday, date_of_hire;
```

You can declare parallel structures of different shapes. For example:

```
struct date birthday:employees, date_of_purch:equipment;
```

Note the different syntax, with "`:`*shapename*" coming after each parallel variable.

You can also use this syntax for declaring a parallel structure:

```
struct date {
    int month;
    int day;
```

```
    int year;
    }:employees birthday;
```

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

Some additional points about structures:

- Only scalar (that is, non-parallel) variables are allowed within parallel or scalar structures. Pointers to parallel variables are allowed within scalar structures, however.
- Shapes are not allowed within parallel or scalar structures; a pointer to a shape is allowed within a scalar structure. (Pointers to shapes are discussed in Chapter 7.)
- You can include a scalar array within a parallel structure; you cannot include pointers of any kind.
- C*, like Standard C, allows structures to be nested.

3.9 Declaring a Parallel Array

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

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

declares an array of three parallel **ints** of shape **employees**, as shown in Figure 13. **ratings[0]** specifies the first of these parallel variables, **ratings[1]** the second, and **ratings[2]** the third.

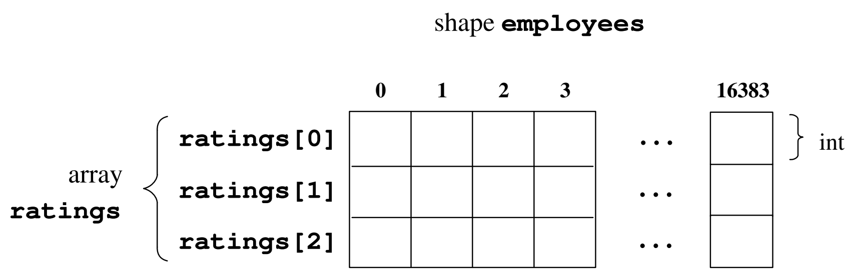

Figure 13. A parallel array of shape `employees`.

Please note the difference between an *element of a parallel array* and an *element of a parallel variable*:

- An element of a parallel array, like **`ratings[0]`** in Figure 13, is a parallel variable. It has values for each position of its shape.
- An element of a parallel variable is scalar, and exists in only one position. **`ratings[0]`** consists of 16384 separate parallel variable elements.

You can also use the alternative syntax for declaring a parallel array. For example:

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

We discuss parallel arrays further in Chapter 7, where we explain their relationship to pointers.

3.10 Initializing Parallel Variables

You can initialize a parallel variable when you declare it. The initializer must be a single scalar value. Each element of the parallel variable is set to that value. For example,

```
shape [65536]ShapeA;
int:ShapeA p1 = 6;
```

sets each element of parallel variable **p1** to 6.

If the variable is an automatic, the initializer can be an expression that can be evaluated at the variable's scope. For example,

```
main()
{
    int i = 12;
    shape [65536]ShapeA;
    int:ShapeA p1 = (6+i);
}
```

sets each element of **p1** to 18.

If there is no initializer in a parallel variable declaration, and the variable has static storage duration, each element of the parallel variable is set to 0; this follows Standard C.

3.10.1 Initializing Parallel Structures and Parallel Arrays

Members of parallel structures and elements of parallel arrays can be initialized only to scalar constants; this too follows Standard C.

3.11 Obtaining Information about Parallel Variables

Once you have declared a parallel variable in a program, you can obtain information about it, just as you can for a shape.

3.11.1 The positionsof, rankof, and dimof Intrinsic Functions

The **positionsof**, **rankof**, and **dimof** intrinsic functions described in Section 3.4 can be applied to parallel variables as well as to shapes. For example, if **age** is a parallel variable of shape **employees**:

- **`rankof(age)`** returns the rank of **`employees`**.
- **`positionsof(age)`** returns the total number of elements of **`age`** (and any other parallel variable of shape **`employees`**). Note that the number of elements of a parallel variable is the same as the number of positions in the parallel variable's shape.
- **`dimof(age,0)`** returns the number of elements in axis 0 of **`age`** (and any other parallel variable of shape **`employees`**).

3.11.2 The shapeof Intrinsic Function

C* includes another intrinsic function that applies only to a parallel variable. The **`shapeof`** intrinsic function takes a parallel variable as an argument and returns the shape of the parallel variable. For example, if a program contains these declarations:

```
shape [16384]employees;
unsigned int:employees age;
```

`shapeof(age)` returns the shape **`employees`**.

`shapeof(age)` is a *shape-valued expression*; it can be used anywhere the shape name **`employees`** is used. For example, once **`age`** is declared, a subsequent declaration of a parallel variable:

```
unsigned int:employees salary;
```

could also be written:

```
unsigned int:shapeof(age) salary;
```

Similarly, a parallel structure like the one shown in Section 3.8 could be declared as follows:

```
struct date:shapeof(age) birthday;
```

3.12 Choosing an Individual Element of a Parallel Variable

As we described earlier, an individual position can be described by its coordinates along the axes of the shape. These coordinates are also used in specifying an individual element of a parallel variable. As with a shape declaration, the coordinates appear in brackets to the left of the variable name, starting with the coordinate for axis 0. These coordinates are also referred to as a *left index*.

Thus, if **age** is a parallel variable of a 1-dimensional shape named **employees**, **[0]age** specifies the first element of **age**, and **[4]age** specifies the fifth element of **age**.

For a 2-dimensional parallel variable called **pvar**,

- **[0][0]pvar** specifies the element in row 0, column 0.
- **[1][0]pvar** specifies the element in row 1, column 0.
- **[0][1]pvar** specifies the element in row 0, column 1.

and so on. Recall that axis 0 refers to the rows, and axis 1 refers to the columns.

A left index must be 0 or greater. The behavior of an operation that includes a negative left index is undefined.

You can use a left index with an element of a parallel array. For example,

```
[77]A1[4]
```

specifies the seventy-eighth parallel variable element of **A1[4]**, which is the fifth array element of the parallel array **A1**.

You can use scalar variables or expressions in place of numbers in the left index. For example, if a program contains this declaration,

```
int j = 4;
```

the expression **[j]age** specifies the fifth instance of **age**.

It is also possible to use parallel variables or expressions in the left index. We leave that topic, however, for Chapter 10.

3.12.1 Precedence

The precedence level of left indexes lies between the bottom of the list of postfix expressions and the top of the list of unary operators.