Chapter 8

Functions

C* adds support for parallel variables and shapes to Standard C functions. Specifically:

- C* functions can take parallel variables and shapes as arguments.
- C* functions can return parallel variables and shapes.
- C* adds a new keyword **`current`**, which you can use to specify that a variable is of the current shape.
- C* includes a **`void`** predeclared shape name so that you can declare an argument to be a pointer to a parallel variable of any shape.
- C* supports overloading of functions, so that (for example) functions operating on scalar and on parallel data can have the same name.

8.1 Using Parallel Variables with Functions

8.1.1 Passing a Parallel Variable as an Argument

C* functions accept parallel variables as arguments only if they are of the current shape. As in Standard C, variables are passed by value; but see Section 8.2 for a discussion of passing by value versus passing by reference.

The simple function below takes a parallel variable of type **`int`** and shape **`ShapeA`** as an argument:

```
void print_sum(int:ShapeA x)
{
    printf ("The sum is %d.\n", +=x);
}
```

(Note that C* supports the new Standard C function prototyping, in addition to the older method. The new method is preferred.) There is actually a better way of writing this function; we describe it in Section 8.4.1.

If **p1** is a parallel variable of type **int** and shape **ShapeA**, you could call **print_sum** as follows:

```
print_sum(p1);
```

provided that **ShapeA** is the current shape. If **ShapeA** were not the current shape, passing **p1** to the function would violate the rule that a program can operate only on parallel variables of the current shape.

NOTE: If a function expects a scalar variable and you pass it a parallel variable instead, you receive a compile-time error.

If the Parallel Variable Is Not of the Current Shape

If you want to pass a parallel variable that is not of the current shape to a function, use a pointer to the parallel variable. Note, though, that if the function is to operate on the parallel variable, the function must include its own nested **with** statement, and the parallel variable that is passed must be of that shape. For example:

```
void print_sum(int:ShapeA *x)
{
  with (ShapeA)
     printf ("The sum is %d.\n", +=*x);
}
```

If **p1** is a parallel variable of type **int** and shape **ShapeA**, you could call **print_sum** as follows, no matter what the current shape is:

```
print_sum(&p1);
```

Section 8.4.2 discusses a more general way of passing parallel variables that are not of the current shape.

8.1.2 Returning a Parallel Variable

C* functions can return parallel values. For example, this function:

```
float:ShapeA increment(float:ShapeA x)
{
    return (x + 1.);
}
```

takes as an argument a parallel variable of type **float** and shape **ShapeA**, and returns, for each active element of the variable, the value of the element plus 1. Assuming that **p1** and **p2** are parallel floats of shape **ShapeA**, and **ShapeA** is the current shape, you could call **increment** as follows:

```
p2 = increment(p1);
```

Note that when a function is to return a parallel variable, you must specify both the type and the shape of the variable. The header of the function **increment** could also have been written with the shape after the parameter list:

```
float increment(float:ShapeA x):ShapeA
```

In a Nested Context

Consider a slightly different version of **increment**:

```
float:ShapeA  increment_if_over_5(float:ShapeA x,
          float:ShapeA y)
{
    where (y > 5.)
          return (x + 1.);
}
```

Figure 30 shows some sample results of a call to this new function.

```
with (ShapeA)
    p3 = increment_if_over_5(p1, p2);
```

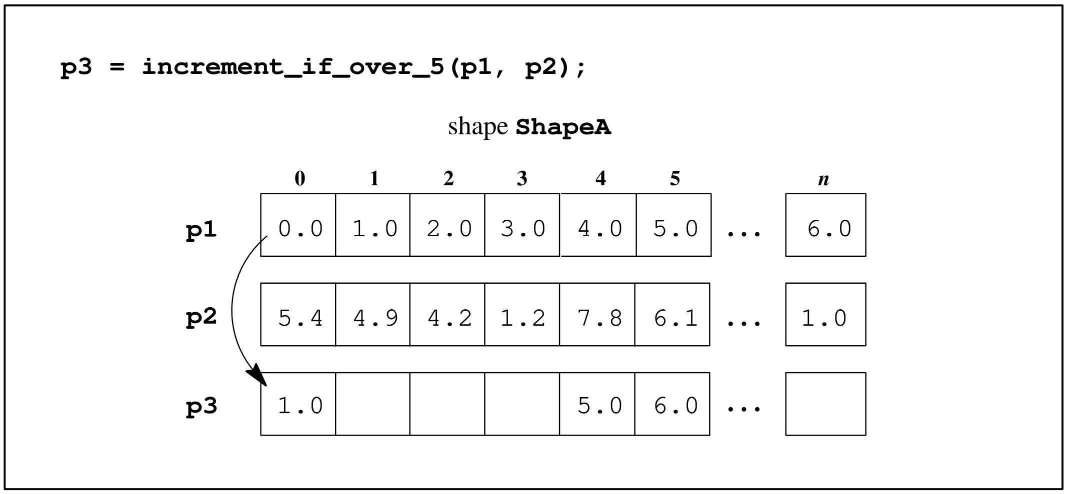

Figure 30. Three parallel variables after a function call.

Upon return from **`increment_if_over_5`**:

- All positions have once again become active, as we discussed in Chapter 6.
- In every position where **`p2`** is greater than 5, the corresponding element of **`p3`** has been assigned the value of the corresponding element of **`p1`** plus 1.
- The values of all other elements of **`p3`** are undefined.

8.2 Passing by Value and Passing by Reference

You can pass parallel variables by value or by reference, just as you can scalar variables. However, in deciding whether to pass by value or pass by reference, you must take into account the effect of inactive positions.

When you pass a variable by value, the compiler makes a copy of it for use in the function. If the variable is parallel, and positions are inactive, elements in those positions have undefined values in the copy. This is not a problem if the function does not operate on the inactive positions; if it does, however, passing by value can produce unexpected results. The function can operate on the inactive positions in these situations:

- If the function contains an **`everywhere`** statement to widen the context, and then operates on the parallel variable you pass.
- If it operates on an individual element of a parallel variable; see Section 6.2.
- If it performs send or get operations involving the parallel variable you pass; send and get operations are described in Chapter 10.

As an example of the first situation, consider this function:

```
float:ShapeA f(float:ShapeA x)
{
    everywhere
        return (8. / x);
}
```

What happens if we pass in a parallel variable with an inactive element? Figure 31 gives an example.

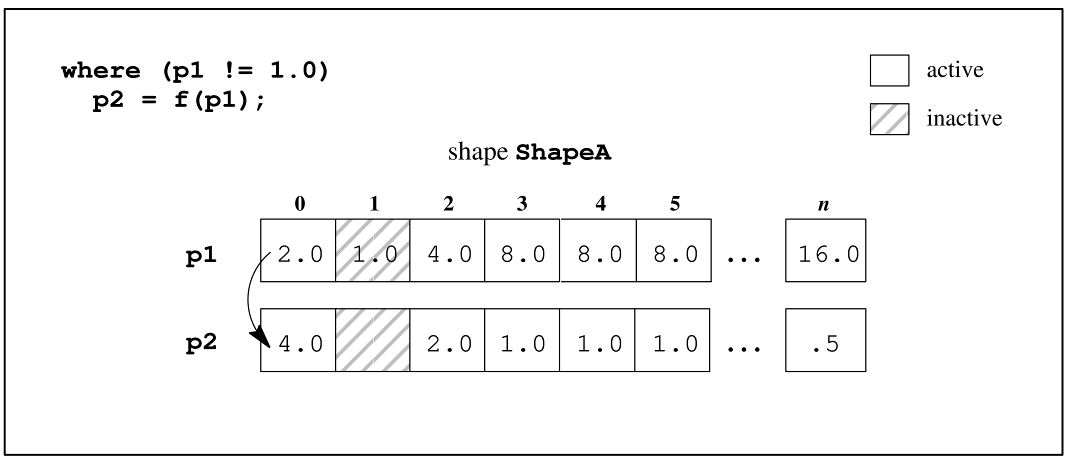

Figure 31. Passing by value when the function contains an `everywhere` statement.

The copy made of **`p1`** contains an undefined value, rather than 1.0, in the inactive position; therefore, the value in **`[1]p2`** is also undefined. Note that you wouldn't want to divide by an undefined value.

To avoid this situation, define the function so that it passes by reference rather than by value.

8.3 Using Shapes with Functions

8.3.1 Passing a Shape as an Argument

C* functions accept shapes as arguments. The function below takes a shape as an argument and allocates a local variable of that shape.

```
int number_of_active_positions(shape x)
{
    with (x) {
         int:x local = 1;
         return (+= local);
    }
}
```

The shape that you pass need not be the current shape.

If the function also returns a parallel variable that is of the shape specified in the parameter list, its shape must be declared *after* the parameter list, to avoid a forward reference. For example:

```
float raise(shape employees, float:employees salary):employees
{
    return (1.1 * salary);
}
```

This format is not especially useful in this case, since **`employees`** must be the current shape. The format becomes more useful when you pass more than one shape, and data is passing between the shapes. For information on communicating between shapes, see the discussion of parallel left indexing in Chapter 10 and the discussion of general communication in Chapter 14.

8.3.2 Returning a Shape

C* functions can also return a shape. For example:

```
shape choose_shape(shape ShapeA, shape ShapeB, int n)
{
    if (n)
         return ShapeA;
    else
         return ShapeB;
}
```

This function returns **`ShapeA`** or **`ShapeB`**, depending on the value of **`n`**.

A function that returns a shape can be used as a shape-valued expression — that is, you can use it in place of a shape name. For example:

```
with (choose_shape(shape1, shape2, s1))
    /* ... */
```

See Section 9.7, however, for limitations on the use of a function as a shape-valued expression when you are declaring a parallel variable.

8.4 When You Don't Know What the Shape Will Be

Some functions you write may be general enough that they can accept a parallel variable of *any* shape as an argument. For example, the **`print_sum`** function used as an example in Section 8.1 could work with any parallel variable. To allow this, C* introduces two new "predeclared" shape names: **`current`** and **`void`**. A predeclared shape name is provided as part of the language; you do not declare it in your program.

8.4.1 The current Predeclared Shape Name

The predeclared shape name **`current`** always equates to the current shape; **`current`** is a new keyword that C* adds to Standard C. You can use **`current`** to declare a parallel variable as follows:

```
int:current variable1;
```

If **`employees`** is the current shape when this statement is executed, **`variable1`** is of shape **`employees`**; if **`image`** is the current shape, **`variable1`** is of shape **`image`**.

NOTE: Since **`current`** is dynamic, you cannot use it with a parallel variable of static storage duration.

Thus, we can generalize **`print_sum`** as follows to let it take any parallel **`int`** of whatever shape is current when the function is called:

```
void print_sum(int:current x)
{
```

```
    printf ("The sum is %d.\n", +=x);
}
```

In fact, this version of the function is more efficient than the version that specifies a particular shape name in the parameter list. If the function specifies a shape name, the compiler has to first make sure that the shape is current, and that the parallel variable is of the current shape. If the function uses **current**, the compiler has to make sure only that the parallel variable is in fact of the current shape.

8.4.2 The void Predeclared Shape Name

C* extends the use of the Standard C keyword **void**. In addition to the standard use, it can be used as the shape modifier for a scalar-to-parallel pointer; it specifies a shape without indicating what the shape's name is. C* does no type checking of a **void** shape.

Use **void** instead of a shape name in a function's parameter list to specify that *any* shape is acceptable as an argument to the function. If you are specifying a parallel variable that can be of any shape, a type specifier (for example, **int**, **float**) is still required. Since you cannot pass a parallel variable that is not of the current shape, **void** must be the shape modifier of a scalar-to-parallel pointer. For example, this function sums the values of the active elements of a parallel **int** of any shape:

```
int sum(int:void *x)
{
    with (shapeof(*x))
        return (+= *x);
}
```

You can also use **void** outside a parameter list to declare a scalar pointer to a parallel variable. For example:

```
int:void *ptr;
```

This declares **ptr** to be a pointer to a parallel **int** of an undetermined shape. The shape is determined by the parallel variable whose address is ultimately assigned to the pointer. For example, if **ptr** points to **p1**:

```
ptr = &p1;
```

then **ptr** is a pointer to an **int** of shape **shapeof(p1)**. But note that a parallel variable of another shape could subsequently be assigned to **ptr**, and the C* compiler would not complain; **ptr** would then simply point to the new parallel variable.

Using shapeof with the void Shape

While convenient, using the **void** shape slows down a program if run-time safety is enabled. It is therefore preferable to use **void** only for the first parameter of a function. For subsequent parameters of the same shape, use the **shapeof** intrinsic function; **shapeof** provides more information to the compiler, thereby allowing the compiler to generate better code. Also use **shapeof** in the controlling expression of the **with** statement to choose the current shape.

For example:

```
int sum_of_two_vars(int:void *x, int:shapeof(*x) *y)
{
    with (shapeof(*x))
         return (+= (*x + *y));
}
```

For parameters declared locally within the function, use **current**:

```
float average(int:void *x)
{
    with (shapeof(*x)) {
         int:current y = 1;
         return (+=*x / +=y);
    }
}
```

Using void when Returning a Pointer

Consider this function, which is passed a shape and returns a pointer to a parallel variable of that shape:

```
int *f(shape ShapeA):ShapeA      /* This is wrong */
{
    /* ... */
}
```

The shape of the return value must come after the parameter list, to avoid a forward reference. However, C* doesn't allow this alternative syntax for a function returning a pointer. The problem is the same as that discussed in Section 7.3.1; the compiler interprets the return value incorrectly as "a parallel pointer of shape **ShapeA** to a scalar **int**," and parallel-to-scalar pointers do not exist in C*.

Use **void** instead of the shape name for the return value in this situation. For example:

```
int:void *f(shape ShapeA)
{
    /* ... */
}
```

Note that this causes an unavoidable loss of some type-checking, since the compiler cannot check for the correct use of the shape of the variable pointed to.

8.5 Overloading Functions

It may be convenient for you to have more than one version of a function with the same name — for example, one version for scalar data and another for parallel data. This is known as *overloading*. C* allows overloading of functions, provided that the functions differ in the type of at least one of their arguments or in the total number of arguments. For example, these versions of function **f** can be overloaded:

```
void f(int x);
void f(int x, int y);
void f(int:current x);
```

Use the **overload** statement to specify the names of the functions to be overloaded. For example, this statement specifies that there may be more than one version of the **increment** function:

```
overload increment;
```

Put the **overload** statement at the beginning of the file that contains the declarations of the functions. The statement *must* appear before the declaration of the second version of the function, and it must appear in the same relative order with respect to the function declarations in all compilation units. Thus, if it appears first in one compilation unit, it must appear first in all compilation units. If you use a header file for your function declarations, this happens by default.

If you have different versions of more than one function, separate the function names by commas in the **`overload`** statement. For example:

```
overload increment, average;
```

NOTE: The CM-200 and CM-5 implementations of C* restrict the shape of parallel formal parameters you can specify in declaring overloaded functions. Only **`current`** and **`void`** can be used in overloaded function declarations.