Chapter 6

Setting the Context

In Chapter 4, we discussed how to use the **with** statement to select a current shape. Once there is a current shape, a program can perform operations on parallel variables that have been declared to be of that shape.

But what if you want an operation to be performed only on certain elements of a parallel variable? For example, you have a database containing the physical characteristics of a population, and you want to know the average height of people who weigh over 150 pounds.

To do this, specify which positions are *active* by using a **where** statement, which C* has added to Standard C. Code in the body of a **where** statement operates only on elements in active positions. Using **where** to specify active positions is known as *setting the context*.

6.1 The where Statement

When a **with** statement first selects a shape, all positions of that shape are active; code in the body of the **with** statement operates on every element of a parallel variable. A **where** statement selects a subset of these positions to remain active. For example, this code:

```
with (population)
    where (weight > 150.0) {
         /* ... */
    }
```

selects only those positions of shape **population** in which the value of parallel variable **weight** is greater than 150. (This assumes that the elements of **weight** have previously been initialized to some values.) Parallel code in the body of the **where** statement applies only to those positions. Figure 20 shows the effect of the **where** statement.

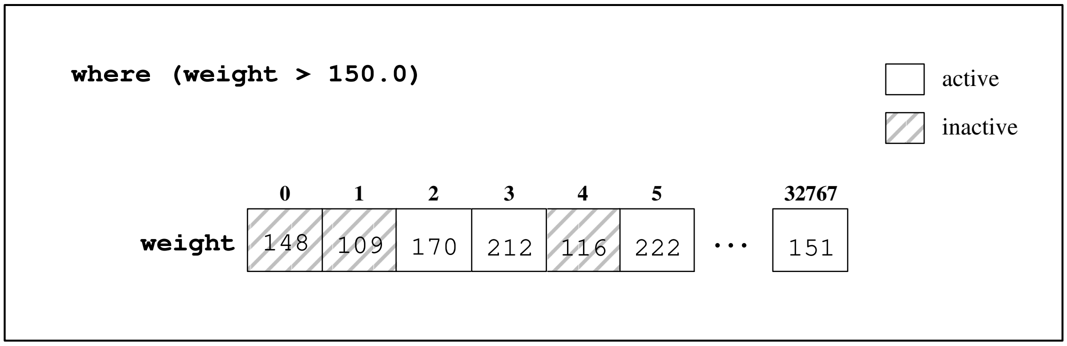

Figure 20. Using where to restrict the context.

In the figure, positions 0, 1, and 4 become inactive in the body of the **where** statement; positions 2, 3, 5, and 32767, all of which have weights over 150, remain active.

The controlling expression that **where** evaluates to set the context must operate on a parallel operand of the current shape. (Other controlling expressions — for example, the **while** and **if** statements — operate only on scalar variables.) Like other controlling expressions, it evaluates to 0 or nonzero, but it does so separately for each parallel variable element that is currently active.

The code below calculates the average height of people weighing over 150 pounds (assuming that the values of **height** and **weight** have been initialized):

```
shape [32768]population;
float:population weight, height;
unsigned int:population count;
float avg_height;

main()
{
    /* Code to initialize height and weight omitted. */

    with (population) {
         count = 1;
         where (weight > 150.0)
              avg_height = (+=height / +=count);
```

```
        }
    }
```

NOTE: There is a slightly easier way of obtaining the number of active positions than the one shown in this code fragment; it involves a *scalar-to-parallel cast*. For example,

```
(int:population)1
```

promotes 1 to a parallel variable of shape **population**. Using the **+=** operator on this variable produces the number of active positions. Scalar-to-parallel casts are discussed in Section 9.6.1.

Like the **with** statement, a **where** statement can include scalar as well as parallel code within its body, and the same restrictions apply to operating on parallel variables that are not of the current shape. See Section 6.5 for a discussion of what happens to scalar and parallel code when a **where** statement causes no positions to remain active.

The context set by the **where** statement remains in effect for any procedures called within its body. Once the body of the **where** statement has been exited, however, the context is reset to what it was before the **where** statement. For example, if we add two statements to the code fragment above:

```
with (population) {
    float avg_weight;
    count = 1;
    where (weight > 150.0)
         avg_height = (+=height / +=count);
    avg_weight = (+=weight / +=count);
}
```

avg_weight is assigned the average weight for all positions of shape **population**, not just for the positions where **weight** is greater than 150.

6.1.1 The else Clause

Like **if** statements in standard C, **where** statements can include an **else** clause. The **else** following an **if** says: *Perform the following operations if the* **if** *condition is not met.* The **else** following a **where** says: *Perform the following operations on positions that were made inactive by the* **where** *condition*. It “turns on” all of the positions that were “turned off” by the **where** condition, and turns off all the positions that the **`where`** condition left on. Figure 21 shows the effect of an **`else`** clause on the set of active positions in Figure 20.

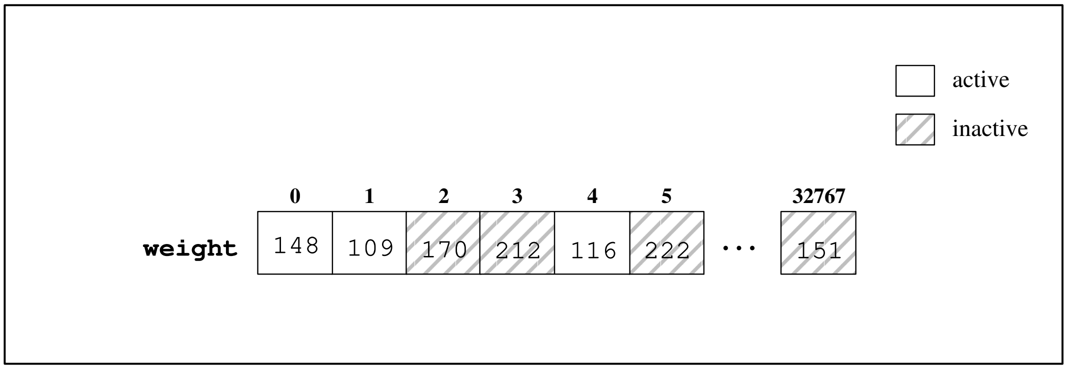

Figure 21. The effect of `else` on the context shown in Figure 20.

The code below calculates separate average heights for those weighing more than 150 pounds, and for those weighing 150 pounds or less:

```
shape [32768]population;
float:population weight, height;
unsigned int:population count;
float avg_height_heavy, avg_height_light;

main()
{
    with (population) {
         count = 1;
         where (weight > 150.0)
              avg_height_heavy = (+=height / +=count);
         else
              avg_height_light = (+=height / +=count);
    }
}
```

6.1.2 The where Statement and positionsof

Using **`where`** to restrict the context does not affect the value returned by the **`positionsof`** intrinsic function. **`positionsof`** returns the total number of positions in a shape, not the number of active positions. See Section 9.6.1 for a method of determining the number of active positions.

6.1.3 The where Statement and Parallel-to-Scalar Assignment

In Chapter 5 we discussed assigning a parallel variable to a scalar variable: you must cast the parallel variable to the type of the scalar variable. The operation then chooses (in an implementation-defined way) one value of the parallel variable and assigns it to the scalar variable. If a **`where`** statement restricts the context, however, the value chosen is from one of the active positions.

6.2 The where Statement and Scalar Code

As we noted above, you can include scalar code within the scope of a **`where`** statement. So, for example, this code is legal:

```
shape [32768]population;
float:population weight;
float avg_height;

main()
{
    with (population) {
         where (weight > 150.0)
              avg_height = 0;
    }
}
```

Recall that an element of a parallel variable is considered to be scalar. That means you can perform operations on an element *even if its position is inactive*. For example, if position 0 becomes inactive when we choose positions where **`weight`** is over 150, we can still do this:

```
shape [32768]population;
float:population weight;
unsigned int:population count;

main()
{
    with (population) {
         count = 1;
         where (weight > 150.0) {
              [0]weight = 225;     /* These are all legal. */
              [0]weight = [1]weight;
              [0]count += count;
         }
```

```
    }
}
```

Note the final statement in this code fragment. In it, the values of the active elements of **`count`** are summed; this sum does not include the value of **`[0]count`**, because position [0] became inactive as a result of the **`where`** statement. However, the result of the sum *can* be placed in **`[0]count`**, because **`[0]count`** is scalar. Thus:

- You can read from or write to an individual parallel variable element in an inactive position.
- An element in an inactive position is not included in operations on the parallel variable as a whole.

6.3 Nesting where and with Statements

6.3.1 Nesting where Statements

You can nest **`where`** statements. The effect is to continually shrink the set of active positions. For example, we might want to calculate average heights separately for males and females weighing over 150 pounds in the **`population`** database. Let's add a parallel variable called **`sex`**, therefore, and assume that it has been initialized: 0 for females and 1 for males. The code below would then produce the desired results.

```
shape [32768]population;
float:population weight, height;
unsigned int:population count, sex;
float avg_male_height, avg_female_height;

main()
{
    with (population) {
         count = 1;
         where (weight > 150.0) {
              where (sex)
                   avg_male_height = (+=height / +=count);
              else
                   avg_female_height = (+=height / +=count);
         }
    }
}
```

6.3.2 Nesting with Statements

It is also possible to choose another shape within the body of a **where** statement. For example:

```
shape [32768]population, [16384]employees;
int:employees salary;
int payroll;
float:population weight, height;
unsigned int:population count, sex;
float avg_male_height, avg_female_height;

main()
{
    with (population) {
         count = 1;
         where (weight > 150.0) {
              where (sex)
                   avg_male_height = (+=height / +=count);
              with (employees)
                   payroll += salary;
         }
    }
}
```

Since each shape has a different set of positions, the context established by a **where** statement for one shape has no effect on the context of expressions in another shape. Therefore, the statement

```
payroll += salary;
```

in the code example above uses the entire set of positions of shape **employees**. Of course, we could add another **where** statement to set the context for the nested **with** statement.

Once control leaves the body of the nested **with** statement, the context returns to whatever it was before the **with** statement was executed. For example:

```
with (population) {
    count = 1;
    where (weight > 150.0)
         where (sex) {
              avg_male_height = (+=height / +=count);
              with (employees)
                   payroll += salary;
         }
         else
```

```
                avg_female_height = (+=height / +=count);
}
```

When **population** becomes the current shape for the second time, the context is once again the positions where **weight** is greater than 150 and **sex** is 0.

With nesting, it is therefore possible to switch back and forth between shapes and maintain separate contexts for each.

6.3.3 The break, goto, continue, and return Statements

Section 4.2 described the behavior of **break**, **goto**, **continue**, and **return** statements in nested **with** statements. They behave similarly for nested **where** statements. Specifically:

- Branching to an outer-level **where** statement resets the context to what it was at that level.
- The behavior of branching into a nested **where** statement is not defined. Don't do it.

The behavior of functions that contain nested **where** statements is discussed in Section 8.1.2.

6.4 The everywhere Statement

A **where** statement can never increase the number of active positions for a given shape; nesting **where** statements has the effect of creating smaller and smaller subsets of the original set of active positions. C* does, however, provide an **everywhere** statement that allows operations on all positions of the current shape, no matter what context has been set by previous **where** statements.

For example, in this code:

```
shape [32768]population;
float:population weight, height;
unsigned int:population count, sex;
float avg_male_height, avg_female_height, avg_height;

main()
```

```
{
    with (population) {
         count = 1;
         where (weight > 150.0) {
              where (sex)
                   avg_male_height = (+=height / +=count);
              else
                   avg_female_height = (+=height / +=count);

              everywhere
                   avg_height = (+=height / +=count);
         }
    }
}
```

the scalar variable **`avg_height`** is assigned the average height for all positions of shape **`population`**, even though this average is calculated within the body of a **`where`** statement that deactivates some positions of **`population`**.

After the **`everywhere`** statement, the context returns to what it was before **`everywhere`** was called. In this case, once again only positions where **`weight`** is greater than 150 are active.

Note that if **`avg_height`** had been calculated after the body of the **`where`** statement, the **`everywhere`** statement would not have been needed, since the context reverts to what it was before the **`where`** statement. In this case, all positions of shape **`population`** become active once again.

As with the **`where`** statement, branching from an **`everywhere`** statement to an outer level via a **`break`**, **`goto`**, **`continue`**, or **`return`** statement resets the context to what it was at the outer level. The behavior of branching into an **`everywhere`** statement is not defined.

6.5 When There Are No Active Positions

What happens when the controlling expression of the **`where`** statement leaves no positions active? Consider the situation shown in Figure 22.

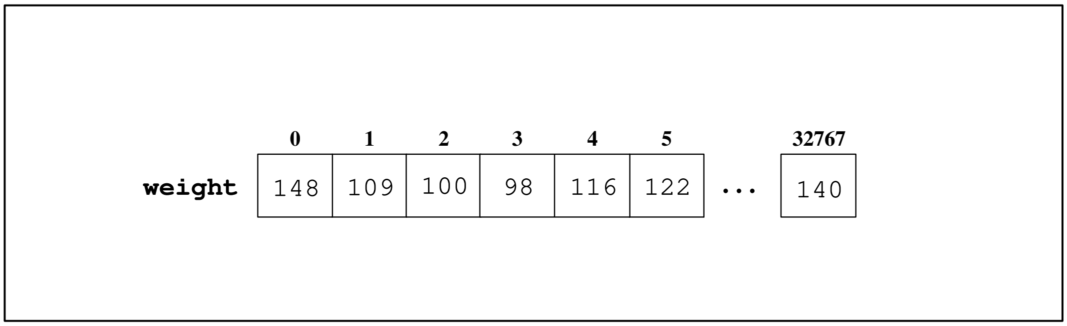

Figure 22. **A shape where all weights are less than 150.**

If **`population`** is initialized entirely with values of 150 and below, the following code makes all positions inactive, since no position has **`weight`** greater than 150:

```
with (population)
     where (weight > 150.0) {
            /* ... */
     }
```

Code is still executed in this situation, but an operation on a parallel variable of the current shape has no effect. For example,

```
weight++;
```

does not increment any of the values of **`weight`**, because no elements of **`weight`** are active.

But note that operations on individual elements do have results, since they are scalar. For example,

```
[0]weight = 225;
```

assigns 225 to element [0] of **`weight`**, even though no positions are active.

The result of a parallel-to-scalar assignment using **=** is undefined when no positions are active.

The results of reduction assignment operations are discussed below.

6.5.1 When There Is a Reduction Assignment Operator

Unary Reduction Operators

Consider the following code fragment, where **maximum** is a scalar variable, and **weight** is a parallel variable:

```
where (weight > 150.0)
    maximum = (>?=weight);
```

If there are no active positions, what gets assigned to **maximum**?

C* provides default values for unary reduction operators when there are no active positions. These values are listed in Table 2.

The values in Table 2 are basically identities for the operations. For example, the result of a **+=** operation (when no positions are active) added to the result of another **+=** operation gives the result of the other operation.

Table 2. Values of unary reduction operators when there are no active positions.

Unary Reduction Operator	Value
+=	0
-=	0
***=**	1
/=	1
&=	~0 (all one bits)
^=	0
|=	0
<?=	maximum value representable
>?=	minimum value representable

Binary Reduction Assignment Operators

Recall that the left-hand side is included in binary reduction assignments. When there are no active positions, and a binary reduction assignment operator is used, the LHS remains unchanged.

6.5.2 Preventing Code from Executing

Of course, you might not want scalar code, or code in another shape, to execute if there are no positions active. To keep the code from executing, use an **if** statement with a bitwise OR reduction operator to conditionalize the entire **where** statement. For example:

```
if (|=(weight > 150.0))
    where (weight > 150.0) {
         float avg_height = 0;
         /* ... */
}
```

In this code fragment, the scalar variable **avg_height** is declared and initialized only if there are any positions with **weight** greater than 150. See Section 5.3.7 for a discussion of using the bitwise OR reduction operator in an **if** condition.

If the condition in the **if** statement has side effects, more code is required to ensure that the condition is evaluated only once. Follow these steps:

1.. Create a temporary parallel variable of the current shape.

2.. In the **if** condition, assign to this temporary variable the results of the parallel expression you would otherwise have evaluated in the **where** statement, and perform a bitwise OR reduction of the temporary variable.

3.. Have **where** evaluate the temporary variable.

For example:

```
with (population) {
    unsigned int:population temporary = 0;
    if (|=(temporary = (++weight > 150.0)))
         where (temporary) {
              float avg_height = 0;
              /* ... */
         }
}
```

6.6 Looping through All Positions

Some of the C* features we have discussed so far can be used to loop through all positions of a shape, allowing operations to be performed on each position separately.

For example, consider a database initialized as shown in Figure 23. Note that each position has a unique identifier, **case_no**.

shape **population**

	0	1	2	3	4	5		32767
case_no	0	1	2	3	4	5	...	32767
weight	148	109	100	212	200	122	...	140
height	62	58	60	72	75	68	...	66

Figure 23. A database.

The code below picks a case of shape **population**, prints the weight and height of its corresponding elements, then picks another case, until all cases have been chosen.

```
#include <stdio.h>

shape [32768]population;
unsigned int:population case_no, weight, height;
unsigned int index;

/* Code to initialize parallel variables omitted. */

main()
{
    with (population) {
          bool:population active;
          active = 1;
          while (|= active) {
                where (active) {
```

```
                        index = (unsigned int)case_no;
                        where (index == case_no) {
                            printf ("Height is %d;
                                   weight is %d.\n",
                                   [index]height, [index]weight);
                            active = 0;
                        }
                    }
                }
            }
        }
```

Note these points about the program:

- In this program, a **while** loop with a bitwise OR reduction controls the selection of positions.
- The **=** operator chooses a value of **case_no** and stores it in **index** (note the use of the cast to explicitly demote the parallel variable to a scalar variable).
- The inner **where** expression then selects the position that contains this value for **case_no**. (There will be only one, because each value of **case_no** is unique.) Since each value of **case_no** corresponds to the coordinate of its position, we can use that value (now assigned to **index**) as a left index for the other parallel variables in order to choose an element of them for printing.
- At the end of the **where** statement, **active** is set to 0 for the active position, turning it off for the next iteration of the loop. When all the positions have been selected, all the positions will have been turned off. At this point the controlling expression of the **while** loop evaluates to false, and the program completes.

NOTE: A more efficient way of doing this is to use the **pcoord** function, which is described in Section 10.2.

6.7 Context and the ||, &&, and ?: Operators

6.7.1 || and &&

The **||** and **&&** operators perform implicit contextualization when one or both of their operands are parallel. (Recall that if one operand is parallel and the other is scalar, the scalar operand is promoted to parallel.)

Consider this statement, in which all variables are parallel:

```
p3 = (p1 > 5) && (p2++);
```

Since at least one of the **&&** operands is parallel, we get the parallel version of the operator. This statement does two things:

- First, in each position, it assigns a 1 to the corresponding element of **p3** if both operands evaluate to nonzero ("TRUE"), and assigns a 0 otherwise.
- Second, it increments **p2** in each position where **p1** is greater than 5 — that is, where the left operand evaluates to TRUE. In positions where the left operand evaluates to 0, **p2** is unchanged.

Figure 24 shows how the statement works with some sample values.

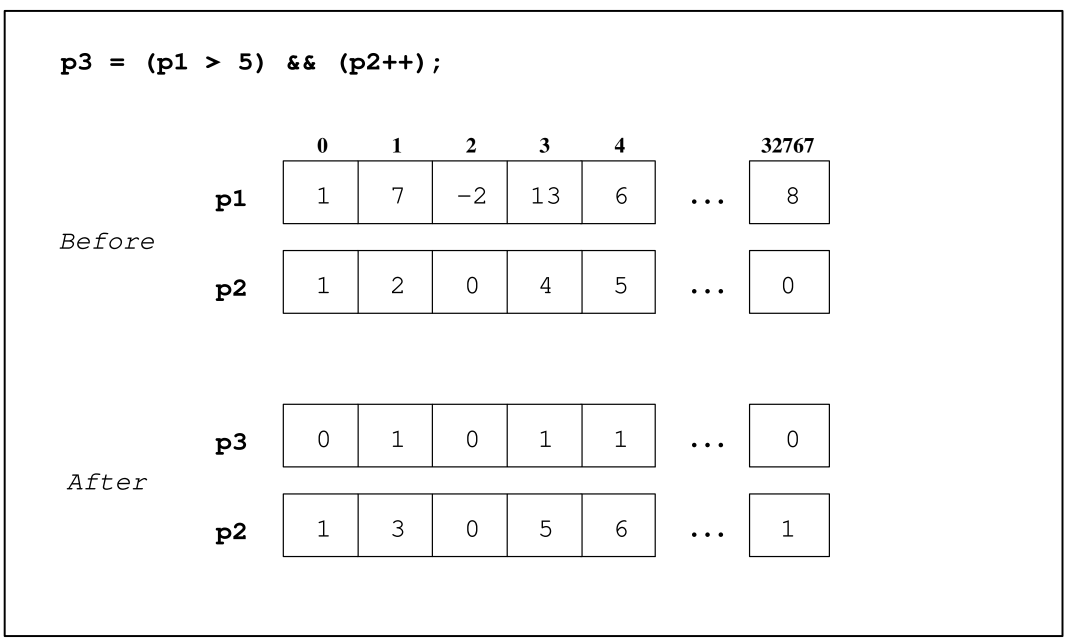

Figure 24. An example of the && operator with parallel operands.

Note that the left operand of the **&&** operator in this example effectively sets the context for the right operand. This is the "implicit contextualization" mentioned at the beginning of the section. That is, the operation above is equivalent to

```
where (p1 > 5)
    p2++;
```

except that the operation additionally returns the result (0 or 1) of the logical AND in each position.

After the operation, the context returns to what it was before the operator was called.

The **||** operator works similarly when one or both of its operands are parallel — except that the context for the right operand consists of those positions that evaluate to 0 for the left operand. In addition, the operator returns a 1 if either operand evaluates to TRUE, and 0 otherwise. For example,

```
p3 = (p1 > 5) || (p2++);
```

gives the results shown in Figure 25.

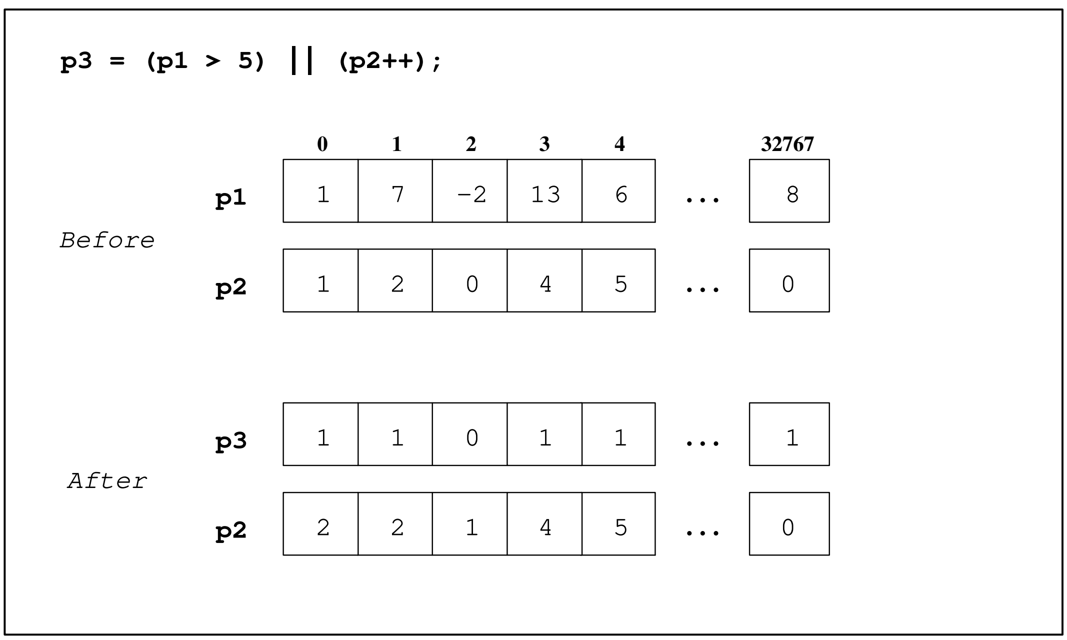

Figure 25. An example of the || operator with parallel operands.

Notice the difference in the results between Figure 24 and Figure 25:

- With the **||** operator, **p2** is incremented only in the positions where **p1** is *not* greater than 5.
- With **||**, the corresponding element of **p3** receives the logical OR of the operands for each position.

6.7.2 The ?: Operator

The **?:** operator provides implicit contextualization of its second and third operands when its first operand is parallel. For example, when **p1** is parallel,

```
(p1 > 5) ? p2++ : p3++;
```

is equivalent to:

```
where (p1 > 5)
    p2++;
else
    p3++;
```

See Section 5.1.5 for an example and for further discussion of this operator.

Appendix A discusses some efficiency considerations for CM-200 C* regarding C* operators that perform implicit contextualization. See the *CM-5 C* Performance Guide* for similar information for CM-5 C*.