Chapter 5

Using C* Operators and Data Types

C* uses all the Standard C operators, plus a few new operators of its own. In addition, C* provides new meanings for the Standard C operators when they are used with parallel variables. Sections 5.1–5.3 of this chapter describe C* operators and how to use them.

C* also provides a new data type, **bool**, which it adds to the Standard C data types. Section 5.4 describes **bool**s.

Section 5.5 discusses parallel unions.

Throughout the chapter, variables beginning with *s* (for example, **s1**, **s2**) are scalar; variables beginning with *p* (**p1**, **p2**) are parallel.

5.1 Standard C Operators

5.1.1 With Scalar Operands

If all the operands in an operation are scalar, C* code performs exactly like Standard C code. Therefore, code like this:

```
int s1=0, s2;
s2 = s1 << 2;
s1++;
s1 += s2;
```

allocates scalar variables and carries out the specified operations on them, just as in Standard C.

The more interesting situations occur when a parallel operand is involved in an operation. The rest of this section considers these situations.

5.1.2 With a Scalar Operand and a Parallel Operand

You can use Standard C binary operators when one of the operands is parallel and one is scalar.

Assignment with a Parallel LHS and a Scalar RHS

We have already shown examples of a parallel left-hand side (LHS) and a scalar right-hand side (RHS) with simple assignment statements, where a scalar constant is assigned to a parallel variable. For example:

```
p1 = 6;
```

In this statement, 6 is assigned to every element of the parallel variable **p1**. Technically, the scalar value is first *promoted* to a parallel value of the shape of the parallel operand, and this parallel value is what is assigned to the elements on the left-hand side.

Similarly,

```
p1 = s1;
```

causes the scalar variable **s1** to be promoted to a parallel variable, and its value is assigned to every element of parallel variable **p1**. Thus, a scalar-to-parallel assignment produces a parallel result; see Figure 14.

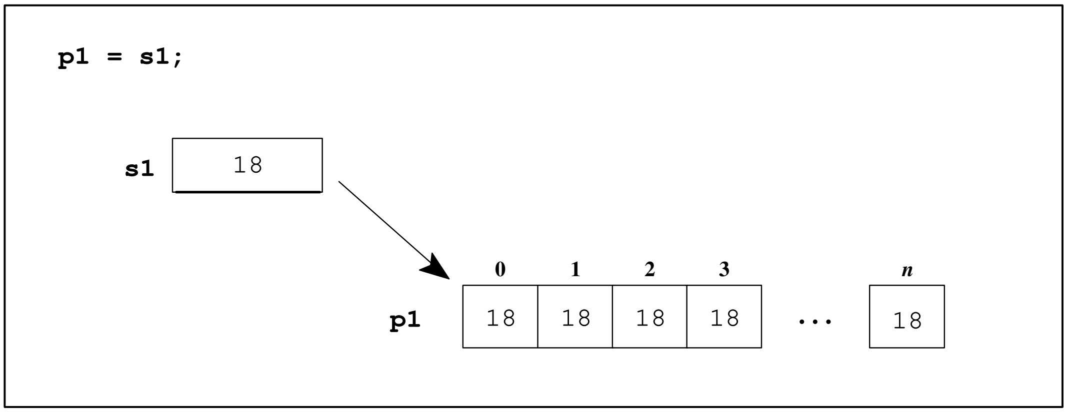

Figure 14. Promotion of a scalar variable to a parallel variable.

Other binary operators work in the same way. For example,

```
p1 + s1
```

adds the value of **s1** to each element of **p1**.

```
p1 == s1
```

tests each element of **p1** for equality to the value of **s1**. For each element, it returns 1 if the values are equal, 0 if they are not equal.

```
p1 << s1
```

shifts the value of each element of **p1** to the left by the number of bits given by the value of **s1**.

```
(p1 > 2) && (s1 == 4)
```

for each element of **p1**, returns 1 if **p1** is greater than 2 and **s1** equals 4; otherwise the expression returns 0 for that element. See Chapter 6 for a further discussion of the **&&** operator when one or both of its operands is parallel.

Assignment with a Scalar LHS and a Parallel RHS

In an assignment statement, promotion occurs only when the scalar variable is on the right-hand side and the parallel variable is on the left-hand side. A scalar variable on the left-hand side is not promoted, and this statement generates a compile-time error:

```
s1 = p1;    /* This is wrong */
```

You can, however, explicitly *demote* the parallel variable to a scalar variable, by casting the parallel variable to the type of the scalar variable. For example:

```
int s1;
int:ShapeA p1;

s1 = (int)p1;  /* This works */
```

(Parallel-to-scalar casts are discussed in more detail in Section 9.6.4.) But what value does C* assign, when the parallel variable could have thousands of different values?

In the case of a simple parallel-to-scalar assignment, with the parallel variable cast to the type of the scalar, C* simply chooses one value of the parallel variable and assigns that value to the scalar variable; see Figure 15. The value that is chosen is defined by the implementation.

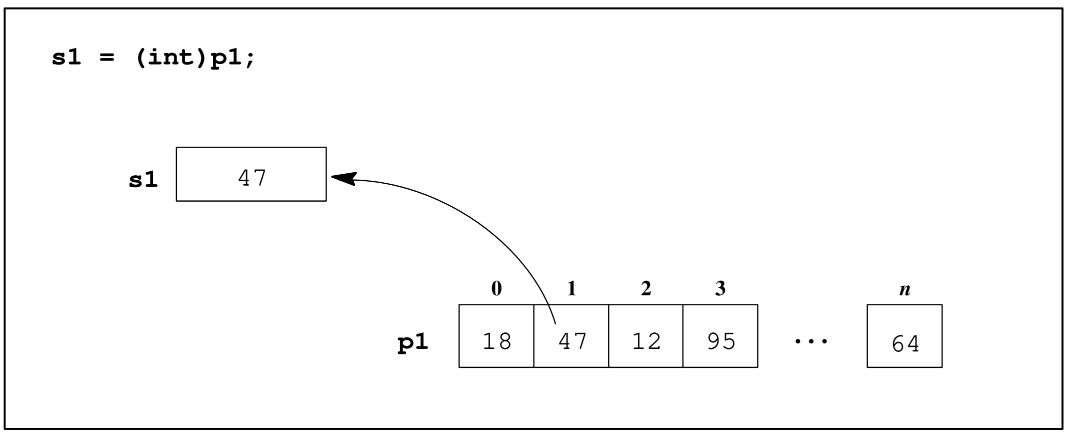

Figure 15. Selection of a value in a parallel-to-scalar assignment.

What is the point of obtaining the value of an element of a parallel variable, if the language doesn't specify which value it will be? One use of demoting a parallel variable to a scalar is to cycle through all elements of a parallel variable and operate on each in turn individually; Chapter 6 has an example of this.

Note that the issues discussed here do not affect a statement like this:

```
s1 = [2]p1;
```

This is a scalar operation. In it, an individual element of **`p1`** has been selected by using the left index **`[2]`**. Since only one element is selected, the value of that element can be assigned to **`s1`** without a problem.

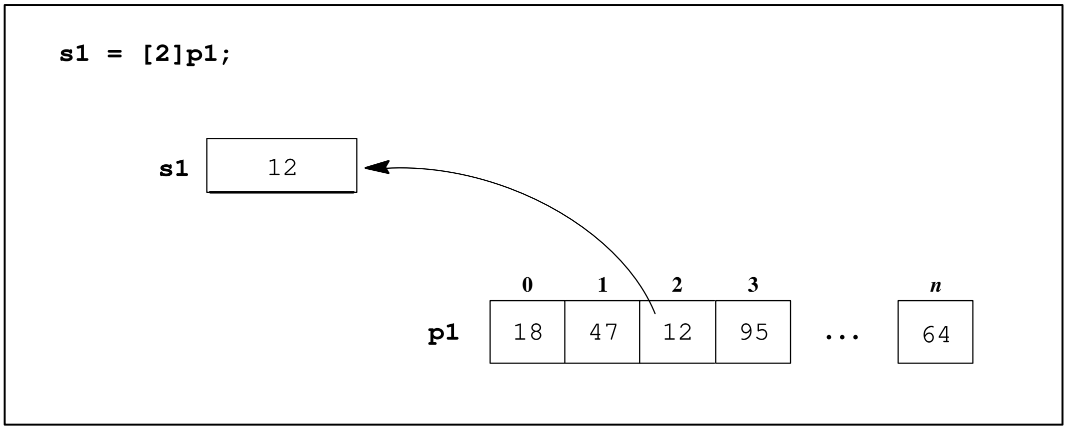

Figure 16. Assignment of a single element of a parallel variable to a scalar variable.

The C compound assignment operators (for example, **`+=`** and **`-=`**) have a special use with a parallel RHS and a scalar LHS; they are discussed in Section 5.3.

5.1.3 With Two Parallel Operands

Standard binary C operators can work with two parallel operands, if both are of the current shape. For example,

```
p2 = p1;
```

assigns the value in each element of **`p1`** to the element of **`p2`** that is at the same position — that is, to the *corresponding element* of **`p2`**; see Figure 17.

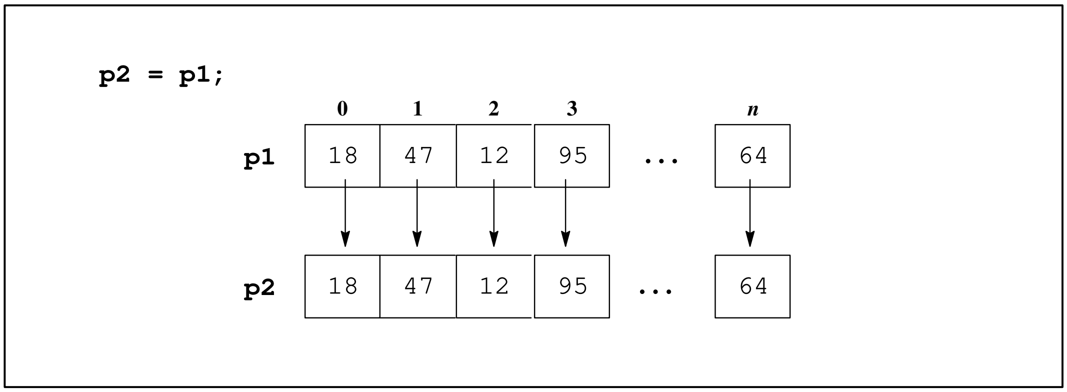

Figure 17. Assignment of a parallel variable to a parallel variable.

```
p1 * p2
```

multiplies each element of **`p1`** by the corresponding element of **`p2`**.

```
p1 >= p2
```

returns, for each element of **`p1`**, 1 if it is greater than or equal to the corresponding element of **`p2`**, and 0 if it is not.

```
(p1 > 2) || (p2 < 4)
```

returns, for each element, 1 if **`p1`** is greater than 2 or **`p2`** is less than 4, and 0 otherwise. Both operands are evaluated if either is parallel. See Section 6.7, however, for a further discussion of this operator and the **`&&`** operator.

5.1.4 Unary Operators for Parallel Variables

Standard C unary operators can be applied to parallel variables. For example,

```
p1++
```

increments the value in every element of the parallel variable **`p1`**.

```
!p1
```

yields the logical negation of each element of **`p1`**. If the value of the element is 0, the expression returns 1; if the value of the element is nonzero, the expression returns 0.

5.1.5 The Conditional Expression

The ternary conditional expression **?:** operates in slightly different ways depending on the mix of parallel and scalar variables in the expression.

For example, in this statement:

```
p1 = (s1 < 5) ? p2 : p3;
```

the first operand is scalar, and the other two operands are parallel. The interpretation of this statement is relatively straightforward: if the scalar variable **s1** is less than 5, the value in each element of the parallel variable **p2** is assigned to the corresponding element of **p1**; if **s1** is 5 or greater, the value in each element of **p3** is assigned to **p1**. All the parallel variables must be of the current shape.

In this statement,

```
p1 = (s1 < 5) ? p2 : s2;
```

the first operand and one of the other operands are scalar. In this case, **s2** is promoted to a parallel variable of the current shape, and the expression is evaluated in the same way as the previous example.

What happens if the first operand is parallel? For example:

```
p1 = (p2 < 5) ? p3 : p4;
```

In this case, each element of **p2** is evaluated separately. If the value in **p2** is less than 5 in a particular element, the value of **p3** is assigned to **p1** for the corresponding element. Otherwise, the value of **p4** is assigned to **p1**. Figure 18 gives an example of this; the arrows in the figure show examples of the data movement, based on the value of **p2**.

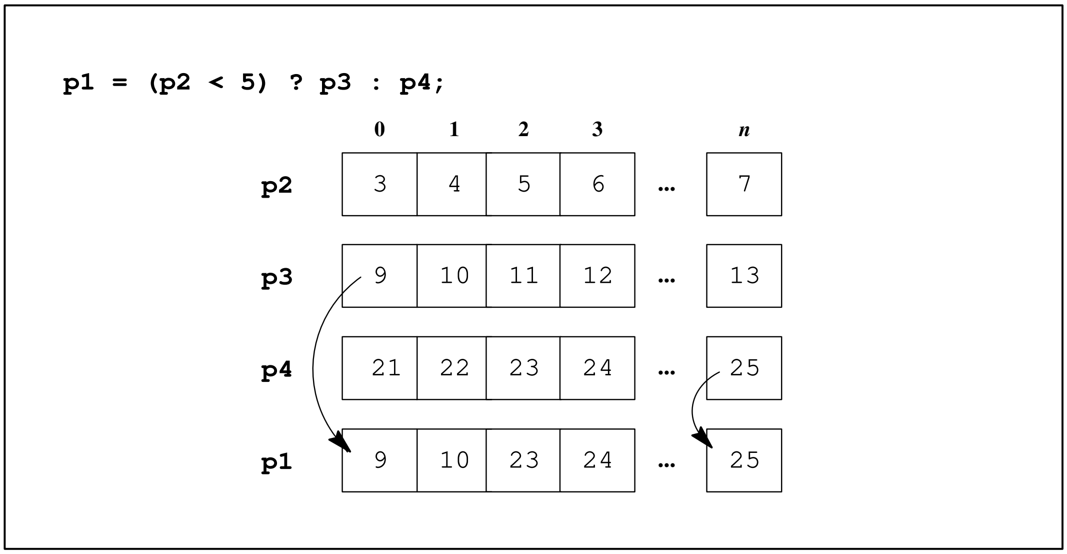

Figure 18. Use of the conditional operator with parallel variables.

If either or both of the operands (other than the first) were scalar in this example, they would be promoted to parallel in the current shape, and the expression would be evaluated in the same way.

Both operands are evaluated if the condition is parallel.

See Section 6.7 for a further discussion of this operator.

5.2 New C* Operators

C* adds several new operators to Standard C.

5.2.1 The <? and >? Operators

The **<?** and **>?** operators provide, respectively, the minimum and maximum of two expressions. These operators are typically expressed as macros in standard C. For example, the C macro

```
( ((a) < (b)) ? (a) : (b) )
```

is similar to

```
a <? b;
```

in C*, except that C* evaluates the operands only once.

There are also assignment operator versions of **<?** and **>?**. For example,

```
s1 >?= s2;
```

assigns the value of **s2** to **s1** if the value is greater than the value of **s1**; otherwise **s1** is unchanged.

The minimum and maximum operators follow Standard C rules for type conversions and compatibility. For example, if one operand is a **float** and the other is an **int**, the **int** is promoted to a **float**. The precedence and associativity of **<?** and **>?** is the same as for the binary relational operators in Standard C.

These operators can be used with parallel as well as scalar variables. For example,

```
p1 <?= p2;
```

assigns the lesser of **p1** and **p2** to **p1**, for every pair of corresponding elements of these parallel variables.

The minimum and maximum operators are discussed further in Section 5.3.

5.2.2 The %% Operator

The new **%%** operator provides the modulus of its operands. It is patterned after the Standard C **%** operator; for example, it has the same precedence and associativity, accepts and returns the same types, and performs the same conversions. It also gives the same answer when both of its operands are positive — the answer is the remainder when the first operand (the numerator) is divided by the second operand (the denominator). For example, these statements are both true:

```
(8 % 6) == 2
(8 %% 6) == 2
```

The difference between the two occurs when one or both of the operands is negative. In that case, different implementations of **%** can give different answers. For example, the sign of the answer can be either positive or negative.

%% performs these steps when one or both of the operands is negative:

1.. It divides the first operand by the second operand. If the result is not an integer, it converts this result to the next lower integer. For example, the result of dividing 17 by –4 is –4.25, so **%%** converts this to –5, because –5 is smaller than –4.

2.. It multiplies the second operand by this result. In the above example, –5 * –4 is 20.

3.. It subtracts *that* result from the first operand. The answer is the result of the operation. In our example, 17 minus 20 is –3. Therefore:

```
(17 %% -4) == -3
```

A consequence of this procedure is that the result always has the same sign as that of the second operand. For example:

```
(-17 %% 4) == 3
(17 %% 4) == 1
(-17 %% -4) == -1
```

The **%%** operator is discussed further in Section 10.3.2.

5.3 Reduction Operators

Standard C has several compound assignment operators, such as **+=**, that perform a binary operation and assign the result to the LHS. Many of these operators can be used with parallel variables in C* to perform reductions — that is, they *reduce* the values of all elements of a parallel variable to a single scalar value. C* reduction operators provide a quick way of performing operations on all elements of a parallel variable.

The code below presents a parallel-to-scalar reduction assignment.

```
#include <stdio.h>

shape [16384]employees;
unsigned int:employees salary;

main()
{
```

```
   unsigned int payroll=0;

/* Initialization of salary omitted */

    with (employees)
         payroll += salary;

    printf ("Total payroll is $%d.\n", payroll);
}
```

In this code, the **+=** operator sums the value in each element of **salary** and adds this sum to the scalar variable **payroll**, as shown in Figure 19. Note that the scalar variable on the left-hand side is included in the operation; that is why **payroll** must be initialized to 0.

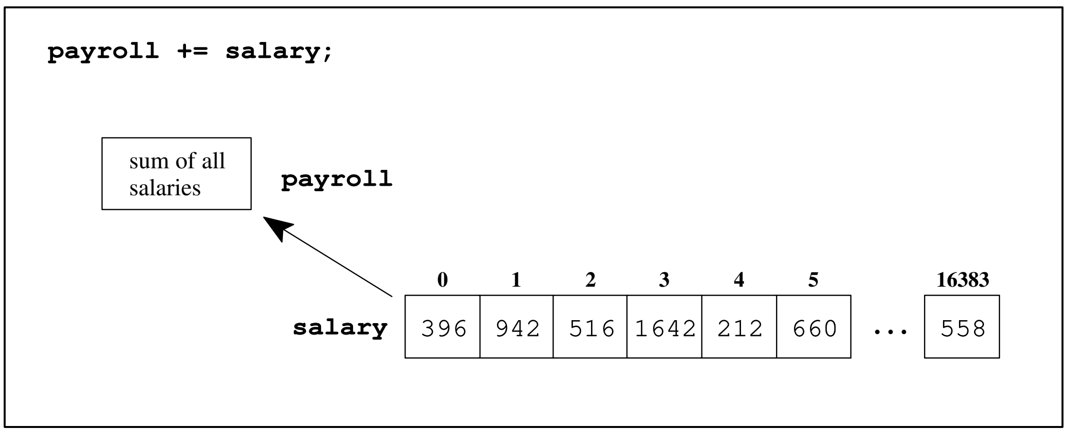

Figure 19. A reduction assignment.

5.3.1 Unary Reduction

As the sample code above shows, binary reduction operators include the left-hand side as one of their operands, so you must initialize the variable on the left-hand side appropriately. You can also use any of these operators as a unary operator with a parallel operand. We can therefore simplify the sample code by eliminating the scalar variable and revising the **printf** statement as follows:

```
printf("Total weekly payroll is $%d.\n", +=salary);
```

5.3.2 Parallel-to-Parallel Reduction Assignment

The left-hand side of a reduction assignment can be an individual element of a parallel variable, instead of a scalar variable. For example,

```
shape [16384]employees;
unsigned int:employees salary, payroll=0;

main()
{
    /* Initialization of salary omitted */

    with (employees)
         [0]payroll += salary;
}
```

declares **payroll** to be a parallel variable, and puts the total of the **salary** values into element [0] of **payroll**.

5.3.3 List of Reduction Operators

Table 1 lists the C* reduction operators. All can be used for parallel-to-scalar reduction assignment, parallel-to-parallel reduction assignment, and unary reduction.

Table 1. Reduction operators.

Operator	Meaning
+=	Sum of values of parallel variable elements
–=	Negative of the sum of values
***=**	Product of values (CM-5 C* only)
/=	Reciprocal of the product of values (CM-5 C* only)
&=	Bitwise AND of values
^=	Bitwise XOR of values
|=	Bitwise OR of values
<?=	Minimum of values
>?=	Maximum of values

Note that simple parallel-to-scalar assignment using a cast is also a form of reduction assignment; see Section 5.1.2.

Note also that the C compound operators **%=**, **<<=**, and **>>=** cannot be used as C* reduction assignment operators.

We have already discussed the **+=** operator; now let's look at the other reduction operators.

5.3.4 The –= Reduction Operator

When used as a binary reduction operator, **-=** subtracts the sum of the parallel RHS's values from the scalar LHS, and assigns the result to the LHS. Therefore,

```
s1 -= p1;
```

is equivalent to:

```
s1 = (s1 - (+=p1));
```

Initialize the scalar LHS to 0 to obtain the negative of the sum of the parallel variable's values. Or use **-=** as a unary reduction operator:

```
s1 = (-=p1);
```

5.3.5 The *= and /= Reduction Operators (CM-5 C* Only)

When used as a binary reduction operator, ***=** multiplies the values of the elements of the parallel RHS and the value of the scalar LHS and assigns the value to the LHS. As a unary operator, it returns the product of the active elements of the parallel variable.

As a binary reduction operator, **/=** divides the value of the scalar LHS by the product of the parallel RHS's values and assigns this value to the LHS. When **/=** is used as a unary operator, it returns the reciprocal of the product of the active parallel values.

These operators are not available in the CM-200 implementation of C*.

5.3.6 Minimum and Maximum Reduction Operators

The **<?=** and **>?=** operators can be used as unary operators to obtain the minimum and maximum values in all elements of a parallel variable. To find out the lowest and highest salaries in the parallel variable **salary**, for example, add these **printf** statements to the code example shown on page 52:

```
printf ("The lowest salary is $%d.\n", <?=salary);
printf ("The highest salary is $%d.\n", >?=salary);
```

Note once again that, when used as binary operators, **<?=** and **>?=** include the left-hand side as an operator. To assign the lowest value of a parallel variable to a scalar variable, therefore,

```
s1 <?= p1;
```

might not work, since **s1** might be the lowest value. Instead, use **<?=** as a unary operator, and use **=** to assign the result to the scalar variable. For example:

```
s1 = <?=p1;
```

5.3.7 Bitwise Reduction Operators

The bitwise reduction aoperators mask all elements of a parallel variable, as described in the subsections below.

Bitwise OR

The **|=** operator performs a bitwise OR of all elements of a parallel variable. For example, in this statement:

```
s1 |= p1;
```

all elements of **p1** are first bitwise OR'd; if a particular bit is a 1 in any element, that bit is a 1 in the result. This result is then bitwise OR'd with **s1**, and the result is assigned to **s1**.

Bitwise OR is particularly useful in testing if any elements of a parallel variable meet a condition. The **if** statement in C* works in the same way as the **if** statement in Standard C: If the condition expression evaluates to 0, then the statement following is not executed; if the condition expression is nonzero, the statement is executed. In this code,

```
if (|=(p1 > 5))
    p2 = 10;
```

if there are any elements of **p1** greater than 5, the condition expression is non-zero, and 10 is assigned to each element of **p2**. If there are no elements of **p1** greater than 5, the bitwise OR evaluates to 0, and the following statement is not executed.

Bitwise AND

In a bitwise AND, if a particular bit is a 0 in any element of the specified parallel variable, that bit is a 0 in the result. Bitwise AND provides a way to test whether all elements of a parallel variable meet a condition. In this code:

```
if (&=(p1 > 5))
    p2 = 10;
```

each element of **p2** is set to 10 only if all elements of **p1** have values greater than 5.

Bitwise Exclusive OR

You can view the bitwise exclusive OR operator as operating pair-wise on elements of a parallel variable. For example, if three parallel bit-fields each contain a 1, bitwise exclusive OR first operates on two of them: the two 1 bits yield a 0 bit. This 0 bit is then exclusive OR'd with the remaining 1 bit, and the result is a 1 bit. In general, the result of a bitwise exclusive OR operation is 1 if the corresponding bit is 1 in an *odd* number of elements; it is 0 if the corresponding bit is 1 in an *even* number of elements. Note that in a reduction assignment the scalar LHS is included in this calculation.

5.3.8 Reduction Assignment Operators with a Parallel LHS

Reduction assignment operators can be used with a parallel LHS when the parallel variable is left-indexed with a parallel subscript. This topic is discussed in Section 10.1.5.

5.3.9 Precedence of Reduction Operators

Unary reduction operators have the same precedence as the unary operators in Standard C.

The precedence and associativity of the binary reduction operators is the same as for the compound assignment operators in Standard C.

5.4 The bool Data Type

The **bool** is a new unsigned integral data type in C*. The actual size and alignment of a **bool** are implementation-dependent:

- **In the CM-200 implementation,** a parallel **bool** occupies one bit of memory and is aligned on a bit boundary; a scalar **bool** occupies one byte of memory on the front end. This takes advantage of the CM-200's alignment of data on bit, rather than byte, boundaries.
- **In the CM-5 implementation,** a **bool** occupies one byte of storage, both on the partition manager and on the nodes, and is aligned on a byte boundary.

A **bool** behaves as a single-bit quantity, however, no matter what its actual size is. Typically, **bool**s are used to test conditions.

When you cast an expression of a larger data type to a **bool**, or assign a variable of a larger data type to a **bool**, the expression has logical (rather than arithmetic) behavior. That is, if the value of the larger data type is 0, 0 is the result; if the value is non-zero, 1 is the result. Thus:

```
int i=0, j=4;
printf("%d\n", (bool)i);  /* prints "0" */
printf("%d\n", (bool)j);  /* prints "1" */
```

Also note this behavior:

```
int i, j=1, k=1;
bool:current b;
i = j + k;  /* i=2 */
b = j + k;  /* b=1 */
```

All elements of **b** are assigned the value 1 because the value of the expression **(j + k)** is non-zero.

A **bool**, like a **char**, is promoted to an **int** when used as an operand of many operators. Thus, performing operations on **bool**s sometimes can be slower than performing the same operations on larger data types. The CM-200 implementation, however, avoids this promotion for operations using these operators: **<**, **>**, **<=**, **>=**, **==**, **!=**, **&**, **|**, **^**, **<?**, **>?**, and the assignment versions of the last five. All these operations are performed at the precision of their operands.

5.4.1 The boolsizeof Operator

To obtain the exact size of a variable or data type in units of **bool**s, use the new C* operator **boolsizeof**; **boolsizeof** has the same precedence and associativity as **sizeof**.

With a Parallel Variable or Data Type

When a parallel variable is used as the operand, **boolsizeof** returns the number of parallel **bool**s a single element of the variable occupies in CM memory. For a parallel data type, **boolsizeof** returns the number of parallel **bool**s that must be allocated for a single instance of the data type. For example:

```
boolsizeof(int:ShapeA); /* Size in parallel bools of a
                           parallel int */
```

In CM-200 C*, a parallel **bool** is implemented as a bit; therefore, **boolsizeof** returns 32 for this statement.

In CM-5 C*, a **bool** is implemented as a byte; therefore, **boolsizeof** returns 4.

With a Scalar Variable or Data Type

When a scalar variable is used as the operand, **boolsizeof** returns the number of scalar **bool**s that the variable occupies in memory. Since a scalar **bool** is stored as a byte in both CM-200 and CM-5 C*, **boolsizeof** gives the same result as the **sizeof** operator for both implementations when applied to a scalar operand. For example,

```
boolsizeof(int);  /* Size in scalar bools of a scalar int */
```

returns 4 for both implementations.

5.5 Parallel Unions

You can create parallel unions. Like parallel structures, they can only contain scalar variables. For example, this code:

```
union ptype {
         int i;
         float f;
};

union ptype:ShapeA p1;
```

defines a parallel variable **`p1`** of shape **`ShapeA`** and of the union type **`ptype`**. This statement initializes **`p1`** as an integer:

```
p1.i = 50;
```

Each element of **`p1`** is an **`int`** containing the value 50.

This statement initializes **`p1`** as a **`float`** containing the value 89.7:

```
p1.f = 89.7;
```

Unions can also appear within structures, as in Standard C.

5.5.1 Limitations

In CM-200 C*, you cannot use an initializer to initialize a parallel union or any object containing a parallel union.

In addition, the following are language restrictions:

- You cannot assign a scalar union to a parallel union.
- You cannot promote a scalar union to be parallel (for example, by a scalar-to-parallel cast; see Section 9.6).

- You cannot demote a parallel union to be scalar.

These restrictions are not present in CM-5 C*, but taking advantage of this will make your program nonportable.

5.6 Parallel Enumeration Type (CM-5 C* Only)

CM-5 C* supports parallel **enum**s. For example,

```
enum color { red, blue, green };
enum color:ShapeA parallel_color;
```

declares the parallel variable **parallel_color** to be of the enumeration type **color**. You can then assign a value to **parallel_color** as follows:

```
parallel_color = red;
```

This assigns the value **red** to every element of the parallel variable **parallel_color**.