parentheses, the macros will sometimes give unexpected—and undesirable—results.

There are two rules to follow when deciding where to put parentheses in a macro definition. First, if the macro's replacement list contains an operator, always enclose the replacement list in parentheses:

```
#define TWO_PI (2*3.14159)
```

Second, if the macro has parameters, put parentheses around each parameter every time it appears in the replacement list:

```
#define SCALE(x) ((x)*10)
```

Without the parentheses, we can't guarantee that the compiler will treat replacement lists and arguments as whole expressions. The compiler may apply the rules of operator precedence and associativity in ways that we didn't anticipate.

To illustrate the importance of putting parentheses around a macro's replacement list, consider the following macro definition, in which the parentheses are missing:

```
#define TWO_PI 2*3.14159
  /* needs parentheses around replacement list */
```

During preprocessing, the statement

```
conversion_factor = 360/TWO_PI;
```

becomes

```
conversion_factor = 360/2*3.14159;
```

The division will be performed before the multiplication, yielding a result different from the one intended.

Putting parentheses around the replacement list isn't enough if the macro has parameters—each occurrence of a parameter needs parentheses as well. For example, suppose that SCALE is defined as follows:

```
#define SCALE(x) (x*10)    /* needs parentheses around x */
```

During preprocessing, the statement

```
j = SCALE(i+1);
```

becomes

```
j = (i+1*10);
```

Since multiplication takes precedence over addition, this statement is equivalent to

```
j = i+10;
```

Of course, what we wanted was

```
j = (i+1)*10;
```