The only remaining operators are = and +=. Both operators are adjacent to b, so we must take associativity into account. Assignment operators group from right to left, so parentheses go around the += expression first, then the = expression:

```
(a = (b += (((c++) - d) + ((--e) / (-f)))))
```

The expression is now fully parenthesized.

Order of Subexpression Evaluation

The rules of operator precedence and associativity allow us to break any C expression into subexpressions—to determine uniquely where the parentheses would go if the expression were fully parenthesized. Paradoxically, these rules don't always allow us to determine the value of the expression, which may depend on the order in which its subexpressions are evaluated.

logical *and* and *or* operators ➤*5.1*
conditional operator ➤*5.2*
comma operator ➤*6.3*

C doesn't define the order in which subexpressions are evaluated (with the exception of subexpressions involving the logical *and*, logical *or*, conditional, and comma operators). Thus, in the expression (a + b) * (c - d) we don't know whether (a + b) will be evaluated before (c - d).

Most expressions have the same value regardless of the order in which their subexpressions are evaluated. However, this may not be true when a subexpression modifies one of its operands. Consider the following example:

```
a = 5;
c = (b = a + 2) - (a = 1);
```

The effect of executing the second statement is undefined; the C standard doesn't say what will happen. With most compilers, the value of c will be either 6 or 2. If the subexpression (b = a + 2) is evaluated first, b is assigned the value 7 and c is assigned 6. But if (a = 1) is evaluated first, b is assigned 3 and c is assigned 2.

Avoid writing expressions that access the value of a variable and also modify the variable elsewhere in the expression. The expression (b = a + 2) - (a = 1) accesses the value of a (in order to compute a + 2) and also modifies the value of a (by assigning it 1). Some compilers may produce a warning message such as "*operation on 'a' may be undefined*" when they encounter such an expression.

To prevent problems, it's a good idea to avoid using the assignment operators in subexpressions; instead, use a series of separate assignments. For example, the statements above could be rewritten as

```
a = 5;
b = a + 2;
a = 1;
c = b - a;
```

The value of c will always be 6 after these statements are executed.