calculates a new value, but it doesn't change the value of \( x \). But evaluating \( ++x + 15 \) does have a side effect, because it involves incrementing \( x \).

From expression to statement is a short step; just add a semicolon. Thus

\[
age = 100
\]

is an expression, whereas

\[
age = 100;
\]

is a statement. Any expression can become a statement if you add a semicolon, but the result might not make programming sense. For example, if \( \text{rodents} \) is a variable, then

\[
\text{rodents} + 6; \quad // \text{valid, but useless, statement}
\]

is a valid C++ statement. The compiler allows it, but the statement doesn't accomplish anything useful. The program merely calculates the sum, does nothing with it, and goes on to the next statement. (A smart compiler might even skip the statement.)

**Nonexpressions and Statements**

Some concepts, such as knowing the structure of a for loop, are crucial to understanding C++. But there also are relatively minor aspects of syntax that suddenly can bedevil you just when you think you understand the language. We'll look at a couple of them now.

Although it is true that adding a semicolon to any expression makes it a statement, the reverse is not true. That is, removing a semicolon from a statement does not necessarily convert it to an expression. Of the kinds of statements we've used so far, return statements, declaration statements, and for statements don't fit the \( \text{statement} = \text{expression} + \text{semicolon} \) mold. For example, although

\[
\text{int toad};
\]

is a statement, the fragment \( \text{int toad} \) is not an expression and does not have a value. This makes code such as the following invalid:

\[
\text{eggs} = \text{int toad} * 1000; \quad // \text{invalid, not an expression}
\]