Then, the program uses the *loop update* part of the loop to increase \( i \) by 1:

\[ i++ \]

This uses the `++` operator, called the *increment operator*. It increments the value of its operand by 1. (The increment operator is not restricted to *for* loops. For example, you can use

\[ i++; \]

instead of

\[ i = i + 1; \]

as a statement in a program.) Incrementing \( i \) completes the first cycle of the loop.

Next, the loop begins a new cycle by comparing the new \( i \) value with 5. Because the new value (1) also is less than 5, the loop prints another line and then finishes by incrementing \( i \) again. That sets the stage for a fresh cycle of testing, executing a statement, and updating the value of \( i \). The process continues until the loop updates \( i \) to 5. Then, the next test fails, and the program moves on to the next statement after the loop.

### for Loop Parts

A *for* loop, then, provides a step-by-step recipe for performing repeated actions. Let's take a more detailed look at how it's set up. The usual parts of a *for* loop handle these steps:

- Setting a value initially
- Performing a test to see whether the loop should continue
- Executing the loop actions
- Updating value(s) used for the test

The C++ loop design positions these elements so that you can spot them at a glance. The initialization, test, and update actions constitute a three-part control section enclosed in parentheses. Each part is an expression, and semicolons separate the expressions from