for (x = 20; x > 5; x--) // continue while x is greater than 5
for (x = 1; y != x; x++) // continue while y is not equal to x
for (cin >> x; x == 0; cin >> x))    // continue while x is 0

The relational operators have a lower precedence than the arithmetic operators. That means the expression

\[ x + 3 > y - 2 \] // expression 1

corresponds to

\[ (x + 3) > (y - 2) \] // expression 2

and not the following:

\[ x + (3 > y) - 2 \] // expression 3

Because the expression \((3 > y)\) is either 1 or 0 after the bool value is promoted to int, expressions 2 and 3 both are valid. But most of us would want expression 1 to mean expression 2, and that is what C++ does.

**The Mistake You’ll Probably Make**

Don't confuse testing the is-equal-to operator (==) with the assignment operator (=). The expression

\[ \text{musicians} == 4 \] // comparison

asks the musical question, is musicians equal to 4? The expression has the value true or false. The expression

\[ \text{musicians} = 4 \] // assignment

assigns the value 4 to musicians. The whole expression, in this case, has the value 4, because that's the value of the left side.

The flexible design of the for loop creates an interesting opportunity for error. If you accidentally drop an equal sign (=) from the == operator and use an assignment