The Relational Operators

The six relational operators are all binary operators which compare two values and return `true` if the relationship holds between the two, and `false` otherwise. Assume we have these variables:

```
int a = 3, b = 5, c = 2;
string s1 = "sam", s2 = "mary";
```

Here are the six operators. Each condition listed here is `true`.

- **Equals**: `==`. `if (a == b - c) ...`
- **Not-equals**: `!=`. `if (a != b) ...`
- **Less-than**: `<`. `if (s2 < s1) ...`
- **Less-than-or-Equals**: `<=`. `if (a <== b - c) ...`
- **Greater-than**: `>`. `if (s1 > s2) ...`
- **Greater-than-or-Equals**: `>=`. `if (b >== a + c) ...`

Relational operators compare primitive types, but they **also work with many of the types supplied by libraries**, such as `string` and `vector`. Again, this is different than Java, where you have to use `equals()` or `compareTo()` to compare `String` objects.

More Pitfalls

As in Java and Python, the equality (`==`) operator uses **two** `=` symbols.; a single is the **assignment** operator. Unlike those languages, accidentally using a `=` when you mean to use `==` creates an **embedded assignment**, which is legal, not what you expect.

```
if (area = 6) ... // always true
```

This would be a syntax error in Java or Python. In C++ it **assigns** the value `6` to the variable `area`, and then, when the condition is evaluated, converts that `6` to `true`.

Comparing floating-point numbers is legal (syntactically) using the relational operators, but it is also problematic. (This is actually true in any programming language; it's not unique to C++.) For instance, the following expressions evaluate to `false`, **not** `true`, even though they are both mathematically true:

```
1.0 == .1 + .1 + .1 + .1 + .1 + .1 + .1 + .1 + .1 + .1; // false
sqrt(2.0) * sqrt(2.0) == 2.0;  // false
```

These occur becuase of **representational errors** in binary numbers. Just as the number 1/3 can't be exactly represented in base-10 (decimal), many numbers cannot be precisely represented in base-2 (binary). When you do calculations with these numbers, those small errors are magnified, and you end up with nonsensical comparisons such as these.

> *To correctly compare floating-point numbers, you must first calculate the absolute value of the difference between the two numbers, and then compare that to a predetermined limit, called* ***epsilon***.