5.6 — Relational operators and floating point comparisons

ALEX AUGUST 31, 2021

Relational operators are operators that let you compare two values. There are 6 relational operators:

Operator	Symbol	Form	Operation
Greater than	>	x > y	true if x is greater than y, false otherwise
Less than	<	x < y	true if x is less than y, false otherwise
Greater than or equals	>=	x >= y	true if x is greater than or equal to y, false otherwise
Less than or equals	<=	x <= y	true if x is less than or equal to y, false otherwise
Equality	==	x == y	true if x equals y, false otherwise
Inequality	!=	x != y	true if x does not equal y, false otherwise

You have already seen how most of these work, and they are pretty intuitive. Each of these operators evaluates to the boolean value true (1), or false (0).

Here's some sample code using these operators with integers:

```
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";
    int x{};
    std::cin >> x;

    std::cout << "Enter another integer: ";
    int y{};
    std::cin >> y;

    if (x == y)
        std::cout << x << " equals " << y << '\n';
    if (x != y)
        std::cout << x << " does not equal " << y << '\n';
    if (x > y)
        std::cout << x << " is greater than " << y << '\n';
    if (x < y)
        std::cout << x << " is less than " << y << '\n';
    if (x >= y)
        std::cout << x << " is greater than or equal to " << y <<
'\n';
    if (x <= y)
        std::cout << x << " is less than or equal to " << y << '\n';

    return 0;
}
```

And the results from a sample run:

```
Enter an integer: 4
Enter another integer: 5
4 does not equal 5
4 is less than 5
4 is less than or equal to 5
```

These operators are extremely straightforward to use when comparing integers.

Boolean conditional values

By default, conditions in an *if statement* or *conditional operator* (and a few other places) evaluate as Boolean values.

Many new programmers will write statements like this one:

```
if (b1 == true)
...
```

This is redundant, as the `== true` doesn't actually add any value to the condition. Instead, we should write:

```
if (b1)
...
```

Similarly, the following:

```
if (b1 == false)
...
```

is better written as:

```
if (!b1)
...
```

Best practice

Don't add unnecessary == or != to conditions. It makes them harder to read without offering any additional value.

Comparison of floating point values

Consider the following program:

```
#include <iostream>

int main()
{
    double d1{ 100.0 - 99.99 }; // should equal 0.01
    double d2{ 10.0 - 9.99 }; // should equal 0.01

    if (d1 == d2)
        std::cout << "d1 == d2" << '\n';
    else if (d1 > d2)
        std::cout << "d1 > d2" << '\n';
    else if (d1 < d2)
        std::cout << "d1 < d2" << '\n';

    return 0;
}
```

Variables d1 and d2 should both have value *0.01*. But this program prints an unexpected result:

```
d1 > d2
```

If you inspect the value of d1 and d2 in a debugger, you'd likely see that d1 = 0.0100000000000005116 and d2 = 0.0099999999999997868. Both numbers are close to 0.01, but d1 is greater than, and d2 is less than.

If a high level of precision is required, comparing floating point values using any of the relational operators can be dangerous. This is because floating point values are not precise, and small rounding errors in the floating point operands may cause unexpected results. We discussed rounding errors in lesson 4.8 -- Floating point numbers if you need a refresher.

When the less than and greater than operators (<, <=, >, and >=) are used with floating point values, they will usually produce the correct answer (only potentially failing when the operands are almost identical). Because of this, use of these operators with floating point operands can be acceptable, so long as the consequence of getting a wrong answer when the operands are similar is slight.

For example, consider a game (such as Space Invaders) where you want to determine whether two moving objects (such as a missile and an alien) intersect. If the objects are still far apart, these operators will return the correct answer. If the two objects are extremely close together, you might get an answer either way. In such cases, the wrong answer probably wouldn't even be noticed (it would just look like a near miss, or near hit) and the game would continue.

Floating point equality

The equality operators (== and !=) are much more troublesome. Consider operator==, which returns true only if its operands are exactly equal. Because even the smallest rounding error will cause two floating point numbers to not be equal, operator== is at high risk for returning false when a true might be expected. Operator!= has the same kind of problem.

For this reason, use of these operators with floating point operands should be avoided.

> **Warning**
>
> Avoid using operator== and operator!= with floating point operands.

So how can we reasonably compare two floating point operands to see if they are equal?

The most common method of doing floating point equality involves using a function that looks to see if two numbers are *almost* the same. If they are "close enough", then we call them equal. The value used to represent "close enough" is traditionally called epsilon. Epsilon is

generally defined as a small positive number (e.g. 0.00000001, sometimes written 1e-8).

New developers often try to write their own “close enough” function like this:

```
#include <cmath> // for std::abs()

// epsilon is an absolute value
bool isAlmostEqual(double a, double b, double epsilon)
{
    // if the distance between a and b is less than epsilon, then a and b are "close
enough"
    return std::abs(a - b) <= epsilon;
}
```

std::abs() is a function in the <cmath> header that returns the absolute value of its argument. So `std::abs(a - b) <= epsilon` checks if the distance between *a* and *b* is less than whatever epsilon value representing “close enough” was passed in. If *a* and *b* are close enough, the function returns true to indicate they’re equal. Otherwise, it returns false.

While this function can work, it’s not great. An epsilon of *0.00001* is good for inputs around *1.0*, too big for inputs around *0.0000001*, and too small for inputs like *10,000*. This means every time we call this function, we have to pick an epsilon that’s appropriate for our inputs. If we know we’re going to have to scale epsilon in proportion to our inputs, we might as well modify the function to do that for us.

Donald Knuth, a famous computer scientist, suggested the following method in his book “The Art of Computer Programming, Volume II: Seminumerical Algorithms (Addison-Wesley, 1969)”:

```
#include <algorithm> // std::max
#include <cmath> // std::abs

// return true if the difference between a and b is within epsilon percent of the larger of a
and b
bool approximatelyEqual(double a, double b, double epsilon)
{
    return (std::abs(a - b) <= (std::max(std::abs(a), std::abs(b)) * epsilon));
}
```

In this case, instead of epsilon being an absolute number, epsilon is now relative to the magnitude of *a* or *b*.

Let’s examine in more detail how this crazy looking function works. On the left side of the <= operator, `std::abs(a - b)` tells us the distance between *a* and *b* as a positive number.

On the right side of the <= operator, we need to calculate the largest value of “close enough” we’re willing to accept. To do this, the algorithm chooses the larger of *a* and *b* (as a rough indicator of the overall magnitude of the numbers), and then multiplies it by epsilon. In this function, epsilon represents a percentage. For example, if we want to say “close enough” means *a* and *b* are within 1% of the larger of *a* and *b*, we pass in an epsilon of 0.01 (1% = 1/100 = 0.01). The value for epsilon can be adjusted to whatever is most appropriate for the circumstances (e.g. an epsilon of 0.002 means within 0.2%).

To do inequality (!=) instead of equality, simply call this function and use the logical NOT operator (!) to flip the result:

```
if (!approximatelyEqual(a, b, 0.001))
    std::cout << a << " is not equal to " << b <<
'\n';
```

Note that while the approximatelyEqual() function will work for most cases, it is not perfect, especially as the numbers approach zero:

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>

// return true if the difference between a and b is within epsilon percent of the larger of a and b
bool approximatelyEqual(double a, double b, double epsilon)
{
 return (std::abs(a - b) <= (std::max(std::abs(a), std::abs(b)) * epsilon));
}

int main()
{
 // a is really close to 1.0, but has rounding errors, so it's slightly smaller than 1.0
 double a{ 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 };

 // First, let's compare a (almost 1.0) to 1.0.
 std::cout << approximatelyEqual(a, 1.0, 1e-8) << '\n';

 // Second, let's compare a-1.0 (almost 0.0) to 0.0
 std::cout << approximatelyEqual(a-1.0, 0.0, 1e-8) << '\n';
}
```

Perhaps surprisingly, this returns:

```
1
0
```

The second call didn't perform as expected. The math simply breaks down close to zero.

One way to avoid this is to use both an absolute epsilon (as we did in the first approach) and a relative epsilon (as we did in Knuth's approach):

```cpp
// return true if the difference between a and b is less than absEpsilon, or within relEpsilon percent of the larger of a and b
bool approximatelyEqualAbsRel(double a, double b, double absEpsilon, double relEpsilon)
{
    // Check if the numbers are really close -- needed when comparing numbers near zero.
    double diff{ std::abs(a - b) };
    if (diff <= absEpsilon)
        return true;

    // Otherwise fall back to Knuth's algorithm
    return (diff <= (std::max(std::abs(a), std::abs(b)) * relEpsilon));
}
```

In this algorithm, we first check if *a* and *b* are close together in absolute terms, which handles the case where *a* and *b* are both close to zero. The *absEpsilon* parameter should be set to something very small (e.g. 1e-12). If that fails, then we fall back to Knuth's algorithm, using the relative epsilon.

Here's our previous code testing both algorithms:

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>

// return true if the difference between a and b is within epsilon percent of the larger of a and b
bool approximatelyEqual(double a, double b, double epsilon)
{
 return (std::abs(a - b) <= (std::max(std::abs(a), std::abs(b)) * epsilon));
}

bool approximatelyEqualAbsRel(double a, double b, double absEpsilon, double relEpsilon)
{
    // Check if the numbers are really close -- needed when comparing numbers near zero.
    double diff{ std::abs(a - b) };
    if (diff <= absEpsilon)
        return true;

    // Otherwise fall back to Knuth's algorithm
    return (diff <= (std::max(std::abs(a), std::abs(b)) * relEpsilon));
}

int main()
{
    // a is really close to 1.0, but has rounding errors
    double a{ 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 };

    std::cout << approximatelyEqual(a, 1.0, 1e-8) << '\n';     // compare "almost 1.0" to 1.0
    std::cout << approximatelyEqual(a-1.0, 0.0, 1e-8) << '\n'; // compare "almost 0.0" to 0.0
    std::cout << approximatelyEqualAbsRel(a-1.0, 0.0, 1e-12, 1e-8) << '\n'; // compare "almost 0.0" to
0.0
}
```

```
1
0
1
```

You can see that with an appropriately picked*absEpsilon*, approximatelyEqualAbsRel() handles the small inputs correctly.

Comparison of floating point numbers is a difficult topic, and there's no "one size fits all" algorithm that works for every case. However, the approximatelyEqualAbsRel() should be good enough to handle most cases you'll encounter.

Next lesson
5.7 Logical operators

Back to table of contents

Previous lesson
5.5 Comma and conditional operators

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

182 COMMENTS

Newest