3.5 — Relational operators (comparisons)

BY ALEX ON JUNE 15TH, 2007 | LAST MODIFIED BY ALEX ON JULY 29TH, 2017

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 all 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.

Comparison of floating point values

Directly comparing floating point values using any of these operators is dangerous. This is because small rounding errors in the floating point operands may cause unexpected results. We discuss rounding errors in detail in section **2.5 -- floating point numbers**.

Here’s an example of rounding errors causing unexpected results:

```
#include <iostream>

int main()
{
    double d1(100 - 99.99); // should equal 0.01
    double d2(10 - 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;
}
```

This program prints an unexpected result:

```
d1 > d2
```

In the above program, d1 = 0.0100000000000005116 and d2 = 0.0099999999999997868. Both numbers are close to 0.01, but d1 is greater than, and d2 is less than. And neither are equal.

Sometimes the need to do floating point comparisons is unavoidable. In this case, the less than and greater than operators (>, >=, <, and <=) are often used with floating point values as normal. The operators will produce the correct result most of the time, only potentially failing when the two operands are close. Due to the way these operators tend to be used, a wrong result often only has slight consequences. The equality operator is much more troublesome since even the smallest of rounding errors makes it completely inaccurate. Consequently, using operator== or operator!= on floating point numbers is not advised. The most common method of doing floating point equality involves using a function that calculates how close the two values are to each other. If the two numbers 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 number (e.g. 0.0000001).

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

```
#include <cmath> // for fabs()
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 fabs(a - b) <= epsilon;
}
```

fabs() is a function in the <cmath> library that returns the absolute value of its parameter. fabs(a - b) returns the distance between a and b as a positive number. This function 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.

While this works, it’s not great. An epsilon of 0.00001 is good for inputs around 1.0, too big for numbers around 0.0000001, and too small for numbers 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 <cmath>

// 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 fabs(a - b) <= ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}
```

In this case, instead of using epsilon as an absolute number, we're using epsilon as a multiplier, so its effect is relative to our inputs.

Let's examine in more detail how the approximatelyEqual() function works. On the left side of the <= operator, the absolute value of 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 1% (1% = 1/100 = 0.01). The value for epsilon can be adjusted to whatever is most appropriate for the circumstances (e.g. 0.01% = an epsilon of 0.0001). 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 many cases, it is not perfect, especially as the numbers approach zero:

```
#include <iostream>
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):

```
// 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 = fabs(a - b);
    if (diff <= absEpsilon)
        return true;

    // Otherwise fall back to Knuth's algorithm
    return diff <= ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * relEpsilon);
}
```

In this algorithm, we've added a new parameter: absEpsilon. First, we check to see if the distance between a and b is less than our absEpsilon, which should be set at something very small (e.g. 1e-12). This handles the case where a and b are both close to zero. If that fails, then we fall back to Knuth's algorithm.

Here's our previous code testing both algorithms:

```
#include <iostream>
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.

3.6 -- Logical operators

Index

3.4 -- Sizeof, comma, and conditional operators

96 comments to 3.5 — Relational operators (comparisons)

« Older Comments 1 2

DecSco
May 31, 2018 at 10:52 am · Reply

Wouldn't it be possible to use the floating point standard to just compare a specified amount of significant digits (bitwise)? While that may be less intuitive, and possibly not as precise, it could be a much quicker way. Or am I wrong?

nascardriver
May 31, 2018 at 12:52 pm · Reply

Hi DecSco!

In order for this to be more efficient it'd need to be implemented on CPU-level.
If you're doing this through software the overhead by extracting the bits would be higher than the performance gain.

DecSco
May 31, 2018 at 1:47 pm · Reply

I thought if you use bitwise operators, it is in fact on that level - I mean, that's why you'd use C/C++ rather than say Python. Or do you mean you'd need specific hardware for that like FPUs?