6.6 — Relational operators and floating point comparisons

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 calculated floating point values can be problematic

Consider the following program:

```
#include <iostream>

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

    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.

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 them to be slightly smaller or slightly larger than expected. And this can throw off the relational operators.

Related content

We discussed rounding errors in lesson 4.8 -- Floating point numbers.

Floating point less-than and greater-than

When the less-than (<), greater-than (>), less-than-equals (<=), and greater-than-equals (>=) operators are used with floating point values, they will produce a reliable answer in most cases (when the value of the operands is not similar). However, if the operands are almost identical, these operators should be considered unreliable. For example, `d1 > d2` happens to produce `true` in the above example, but could have just as easily produced `false` if the numerical errors had gone a different direction.

If the consequence of getting a wrong answer when the operands are similar is acceptable, then using these operators can be acceptable. This is an application-specific decision.

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 and inequality

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.

```
#include <iostream>

int main()
{
    std::cout << std::boolalpha << (0.3 == 0.2 + 0.1); // prints false

    return 0;
}
```

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

Warning

Avoid using operator== and operator!= to compare floating point values if there is any chance those values have been calculated.

There is one notable exception case to the above: It is okay to compare two floating point literals of the same type when the number of significant digits in each literal does not exceed the minimum precision for that type. Float has a minimum precision of 6 significant digits, and double has a minimum precision of 15 significant digits.

We cover the precision for the different types in lesson 4.8 -- Floating point numbers.

For example, you may occasionally see a function that returns a floating point literal (typically `0.0`, or sometimes `1.0`). In such cases, it is safe to do a direct comparison against the same literal value of the same type:

```
if (someFcn() == 0.0) // okay if someFcn() returns 0.0 as a literal only
    // do something
```

Instead of a literal, we can also compare a const or constexpr floating point variable that was initialized with a literal value:

```
constexpr double gravity { 9.8 };
if (gravity == 9.8) // okay if gravity was initialized with a literal
    // we're on earth
```

It is mostly not safe to compare floating point literals of different types. For example, comparing `9.8f` to `9.8` will return false.

Tip

It is okay to compare two floating point literals of the same type when the number of significant digits in each literal does not exceed the minimum precision for that type. Float has a minimum precision of 6 significant digits, and double has a minimum precision of 15 significant digits.

It is generally not safe to compare floating point literals of different types.

Comparing floating point numbers (advanced / optional reading)

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()

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

std::abs() is a function in the <cmath> header that returns the absolute value of its argument. So `std::abs(a - b) <= absEpsilon` checks if the distance between *a* and *b* is less than or equal to 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*.

As an aside…

If we say any number that is within 0.00001 of another number should be treated as the same number, then:

- 1 and 1.0001 would be different, but 1 and 1.00001 would be the same. That's not unreasonable.
- 0.0000001 and 0.00001 would be the same. That doesn't seem good, as those numbers are two orders of magnitude apart.
- 10000 and 10000.0001 would be different. That also doesn't seem good, as those numbers are barely different given the magnitude of the number.

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 the magnitude of 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> // for std::max
#include <cmath>     // for std::abs

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

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 relEpsilon. In this function, relEpsilon 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 relEpsilon of 0.01 (1% = 1/100 = 0.01). The value for relEpsilon 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 (!approximatelyEqualRel(a, b, 0.001))
    std::cout << a << " is not equal to " << b << '\n';
```

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

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

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

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

    constexpr double relEps { 1e-8 };
    constexpr double absEps { 1e-12 };

    std::cout << std::boolalpha; // print true or false instead of 1 or 0

    // First, let's compare a (almost 1.0) to 1.0.
    std::cout << approximatelyEqualRel(a, 1.0, relEps) << '\n';

    // Second, let's compare a-1.0 (almost 0.0) to 0.0
    std::cout << approximatelyEqualRel(a-1.0, 0.0, relEps) << '\n';

    return 0;
}
```

Perhaps surprisingly, this returns:

```
true
false
```

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 or equal to 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.
    if (std::abs(a - b) <= absEpsilon)
        return true;

    // Otherwise fall back to Knuth's algorithm
    return approximatelyEqualRel(a, 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:

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

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

// Return true if the difference between a and b is less than or equal to 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.
    if (std::abs(a - b) <= absEpsilon)
        return true;

    // Otherwise fall back to Knuth's algorithm
    return approximatelyEqualRel(a, b, relEpsilon);
}

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

    constexpr double relEps { 1e-8 };
    constexpr double absEps { 1e-12 };

    std::cout << std::boolalpha; // print true or false instead of 1 or 0

    std::cout << approximatelyEqualRel(a, 1.0, relEps) << '\n';     // compare
"almost 1.0" to 1.0
    std::cout << approximatelyEqualRel(a-1.0, 0.0, relEps) << '\n'; // compare
"almost 0.0" to 0.0

    std::cout << approximatelyEqualAbsRel(a, 1.0, absEps, relEps) << '\n';     //
compare "almost 1.0" to 1.0
    std::cout << approximatelyEqualAbsRel(a-1.0, 0.0, absEps, relEps) << '\n'; //
compare "almost 0.0" to 0.0

    return 0;
}
```

```
true
false
true
true
```

You can see that 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() function with an absEpsilon of 1e-12 and a relEpsilon of 1e-8 should be good enough to handle most cases you'll encounter.

Making the `approximatelyEqual` functions constexpr Advanced

In C++23, the two `approximatelyEqual` functions can be made constexpr by adding the `constexpr` keyword:

```
// C++23 version
#include <algorithm> // for std::max
#include <cmath>     // for std::abs (constexpr in C++23)

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

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

    // Otherwise fall back to Knuth's algorithm
    return approximatelyEqualRel(a, b, relEpsilon);
}
```

Related content

We cover constexpr functions in lesson 5.8 -- Constexpr and consteval functions.

However, prior to C++23, we run into an issue. If these constexpr function are called in a constant expression, they will fail:

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

    constexpr double relEps { 1e-8 };
    constexpr double absEps { 1e-12 };

    std::cout << std::boolalpha; // print true or false instead of 1 or 0

    constexpr bool same { approximatelyEqualAbsRel(a, 1.0, absEps, relEps) }; //
compile error: must be initialized by a constant expression
    std::cout << same << '\n';

    return 0;
}
```

This is because a constexpr function that is used in a constant expression can't call a non-constexpr function, and `std::abs` wasn't made constexpr until C++23.

This is easy to fix though -- we can just ditch `std::abs` for our own constexpr absolute value implementation.

```
// Prior to C++23 version
#include <algorithm> // for std::max
#include <iostream>

// Our own constexpr implementation of std::abs (for use prior to C++23)
// In C++23, use std::abs
// constAbs() can be called like a normal function, but can handle different types of
values (e.g. int, double, etc...)
template <typename T>
constexpr T constAbs(T x)
{
    return (x < 0 ? -x : x);
}

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

// Return true if the difference between a and b is less than or equal to absEpsilon,
or within relEpsilon percent of the larger of a and b
constexpr bool approximatelyEqualAbsRel(double a, double b, double absEpsilon, double
relEpsilon)
{
    // Check if the numbers are really close -- needed when comparing numbers near
zero.
    if (constAbs(a - b) <= absEpsilon)
        return true;

    // Otherwise fall back to Knuth's algorithm
    return approximatelyEqualRel(a, b, relEpsilon);
}

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

    constexpr double relEps { 1e-8 };
    constexpr double absEps { 1e-12 };

    std::cout << std::boolalpha; // print true or false instead of 1 or 0

    constexpr bool same { approximatelyEqualAbsRel(a, 1.0, absEps, relEps) };
    std::cout << same << '\n';

    return 0;
}
```

For advanced readers

The version of `constAbs()` above is a function template, which allows us to write a single definition that can handle different types of values. We cover function templates in lesson 11.6 -- Function templates.